UNIVERSITÄT DUISBURG ESSEN

Open-Minded

Problem-Based Project Planning in Postmodern Software Engineering

Dissertation by

Ina Wentzlaff

Acknowledgements

To get started, I want to thank all participants of the *Promi*-Workshop series at the Working Group Software Engineering whom have spent their valueable time for reviewing reams of research work including mine.

Special thanks goes to my Doktormutter Prof. Dr. rer. nat. Maritta Heisel for her continued support and patience throughout this dissertation project. *Per aspera ad astra!*

Thanks a lot to Prof. Dr.-Ing. Axel Hunger for establishing the *International Studies in Engineering* program at the University of Duisburg-Essen, home ground of my past scholarly activities.

And last but not least, thanks to Prof. Dr. rer. nat. Michael Goedicke for joining this dissertation project to get it done.

It has been a great pleasure to do research and teaching at this inspiring and open-minded place, and a honor to have worked with you all!

Dissertation

Problem-Based Project Planning in Postmodern Software Engineering

Von der Fakultät für Ingenieurwissenschaften,
Abteilung Informatik und Angewandte Kognitionswissenschaft der
Universität Duisburg-Essen

zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften (Dr. rer. nat.)

genehmigte Dissertation
von

Ina Wentzlaff
aus
Verden an der Aller

1. Gutachterin: Prof. Dr. Maritta Heisel
2. Gutachter: Prof. Dr. Michael Goedicke

Tag der mündlichen Prüfung: 12. November 2021

Eureka!

It is done.

To my soul mates – *Markus & Isabelle*

To my dear kids – *Wim & Niki*

To my amigos – *Uschi, Johannes & Dirk*

In return for keeping me running on
at a sustainable pace.

I love you!

Abstract

Embracing change is in these days the preferred mode of operating software development projects as to meet the pressure of time-to-market. In order to stick to deadline, agile project practices have become the means of first choice, because they implement a time-boxed software delivery process, which anticipates the emergence of requirements. The team is at the heart of each successful agile software project, equipped with all encompassing authority to decide on the scope of requirements to be done within the time available. When it comes to planning the delivery of working software on deadline, agile software development culture demands to *Trust the team*.

Coincidentally or not, this agilist credo ignores the emergence of teams. It conceals the need for the team to know its speed in order to reach consensus on a sound work plan, one which satisfies the software product requirements in the time frame set for the project. *Speed* is team-unique, since it depends on the team members perception of the work volume they can accomplish within a given project timebox. Since different teams have different methods and experiences for justifying their work volume, speed is not comparable among them. This becomes a dilemma on projects with frequent membership turnover and projects at scale calling for a team of teams.

The efficacious planning of time-boxed software projects depends on the team members way to *share the knowledge* of establishing *consistent size estimates* for a *recognizable scope of software product requirements* that they use as the basis for measuring speed. The principle idea followed in this dissertation is to make use of patterns as a common point of reference reusable by individuals and teams for adjusting their know-how and comparing their achievements. On these grounds, it introduces *pre-defined units for reproducibly determining the scope and speed of software projects*. It leverages the intertwining of patterns from problem analysis and solution design, for controlling project progress with respect to satisfied software product requirements. The resulting pattern-based units for requirements size estimating and work planning are integrated to an agile project process framework to bring them into action.

This work contributes to the predictability of software project plans, and thus the *planning for value* delivery, from both the perspective of project management and software engineering. It introduces a shareable, problem-based instrument that operates on reusable pattern practices, for *empowering project teams* to enforce change control in the presence of emerging product requirements or team memberships, and *to trust* their efficacy *by a problem-based speed benchmark*. In this context, the proposed approach sustains the project team in justifying their prioritization of the project plan, made possible by *traceable requirements dependencies*, which reflect the software product life-cycle, and thus are eminent to *stabilize the leeway in decision making* for managing the software project. In addition, it provides the project team with exploratory design alternatives, using *instant models* for recognizable software product requirements, which enable them to *fast-track development* of the software product.

Zusammenfassung

Agile Softwareentwicklung verspricht, den heutigen Markterfordernissen von sich stetig ändernden Produktanforderungen und dem daraus resultierenden Druck immer kürzer werdender Produkteinführungszeiten geeignet Rechnung zu tragen. *Erwarte Veränderungen* lautet eine Maxime agiler Projektmanagementpraxis, *Vertraue dem Team* eine andere. Im Vordergrund steht die termingerechte, in der Regel nur wenige Arbeitstage umfassende Lieferung einsatzbereiter Software. Dabei obliegt es dem Projektteam zu entscheiden, welche Produktanforderungen es innerhalb des zur Verfügung stehenden Projektzeitfensters realisieren wird.

Um einen machbaren Projektplan zu erstellen, der die termingerechte Bereitstellung eines gewissen Umfangs gewünschter Softwarefunktionalität zusichert, muss das bewältigbare Arbeitsvolumen eines Projektteams für ein gegebenes Projektzeitfenster bekannt bzw. die erwartbare *Geschwindigkeit* der Projektdurchführung geeignet abzuschätzen sein. In kleinen, eingespielten Teams, wie sie für agile Softwareentwicklungsprojekte typisch sind, lässt sich ein solches Geschwindigkeitsmaß in der Regel händisch, aber zweckerfüllend erzeugen. Mit zunehmender Projektgröße oder bei volatilen Projektmitgliederkonstellationen, sind diese teambestimmten Leistungswerte jedoch nicht mehr anwendbar sprich vergleichbar, was eine informierte und koordinierende Projektplanung erschwert.

Die wirksame Planung zeitgesteuerter Softwareprojekte hängt entscheidend von dem *geteiltem Wissen* ab, das unterschiedliche Projektteams gleichermaßen dazu nutzen, um die zu realisierenden *Softwareanforderungen* einheitlich zu kategorisieren und deren *funktionalen Umfang* mittels vergleichbaren Maßzahlen *konsistent zu beurteilen*. Deshalb verfolgt diese Dissertation die Idee, *Muster* zur Schaffung kombinierter Softwareentwicklungs- und Softwarebewertungs-*Einheiten* einzusetzen, die von Einzelpersonen und Teams als gemeinsamer Bezugspunkt wiederverwendet werden können, um den Umfang und die Geschwindigkeit von Softwareprojekten *reproduzierbar zu bestimmen*. Um diese musterbasierten Projektplanungseinheiten herzustellen, wird die *Verknüpfung von Problemanalyse und Lösungsentwurf* auf Basis von Mustern innerhalb früher Phasen des Softwareprojektmanagements weiterentwickelt, und durch die *Funktionspunktanalyse als Metrik* für eine standardbezogene Bestimmung von Softwareprodukt- und damit assoziierbarer Projektkennzahlen erweitert.

Wiederverwendbare Projektplanungseinheiten, wie sie diese Dissertation einführt und welche die Softwaretechnik- und Projektmanagement-Sicht auf wiederkehrende Softwareprobleme und deren mögliche Lösungen in sich vereinen, dienen der Anpassbarkeit und Prognosefähigkeit von Softwareprojektplänen sowie der damit zusammenhängenden *Objektivierung erreichbarer Wertschöpfung*. Hierfür wird ein musterbasierter Projektplanungsprozess, mit dessen Hilfe sich ein vordefinierter Umfang gewünschter Softwarefunktionalität nachvollziehbar spezifizieren lässt, in ein agiles Projektprozessrahmenwerk integriert und seine Anwendung und sein Beitrag zur Stabilisierung der Planung von Softwareentwicklungsprojekten anhand mehrer Fallstudien vorgestellt. Projektteams werden auf diese Weise in die Lage versetzt, *flexibel auf Veränderungen zu reagieren* und sozusagen *aus dem Stand heraus Varianten eines realisierbaren Projektplans* zu erzeugen. Die Wirksamkeit gewählter Planungsalternativen wird mittels Funktionspunkten als ein teamübergreifendes da problembezogenes Geschwindigkeitsmaß bemessbar und vergleichbar. Der Softwarelebenszyklus wird zum Aufbau *nachvollziehbarer Anforderungsabhängigkeiten* zwischen den einzelnen Planungseinheiten verwendet. Damit werden diese für die *bedarfsorientierte Priorisierung* des Projektplans systematisch berücksichtigbar und tragen somit zu dem Erfordernis *beschleunigter Softwarelieferung* bei.

Introduction

In the beginning was the crisis

The problem given the name "software crisis" was that improvements in computer hardware outpaced the ability of computer software to make use of it. At this juncture the crisis manifested itself in failed projects that, if at all, delivered a poor quality product. Dijkstra described this situation in his Turing Award lecture 1972 as follows "To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming had become an equally gigantic problem." [78, page 3] The increasing complexity of software development made it more and more difficult to deliver a project product that meets its requirements within the planned cost and time. In response, the software crisis was discussed at the NATO Software Engineering Conferences [181] in 1968 and 1969, which promoted the application of well-defined engineering approaches for the development of software. The software engineering discipline was born to solve the crisis. The progress of this new discipline in the following decades can be divided into eras.

The Pre-Modern Era

From the project point of view, the driving question for success in this era is directed to the software production skills: Does the software work to be used? The authority to answer for this question is assigned to the field of software development. It takes responsibility for managing the "typically [...] error-prone and tedious process" [134, page 18] of translating project expectations from requirements specifications into working software. Developers structure and document their know-how, problem-solving rationale, and decision making that are applied during this process in their programs. But their individual engineering efforts is also hidden in these modules. In this upstream software process, which aims to close the semantic gap between requirements specifications and their sufficient implementation, software development excellence determines project progress. In focus of discipline in this era is the program, and how to build it right. This era can be characterized by engineering approaches that apply a sequenced software process, modular programs, "structured methods and functional decomposition" [134, page 16].

The Modern Era

From the project point of view, the driving question for success in this era is directed to the software production scope: Does the software work for its use? The authority to answer for this question is separated into different perspectives: the one dealing with the formulation of the problem, and assuring that it should be addressed, and the one dealing with the integration of its solution to be delivered. Respectively, the fields of software analysis and software design own specific responsibilities to set up and document a problem-solving plan. It has to be done before handover to subsequent software development activities take place. Analyst and designer hold the interpretative sovereignty over what it takes to build the right software. In this downstream software process, their understanding and decision making determine project progress, which is dependent on a structure of structures. It has to be documented by several, related models, those for the requirements, and those for the computational components of a software architecture, which is planned for satisfying these. In focus of discipline in this era is the plan, the upfront “big picture” of the project product or “blueprint” of the software to be built. It has to be accepted and approved before the start of development. Model- and component-based view points have introduced additional meaning into the software process, and enable automated, reuse-driven software development. But their claim for “universalism requires uniformity” [184, page 366] and its “ideals of perfection” [184, page 364] have become a bottleneck in their application. This era can be characterized by engineering approaches that apply encapsulation, object-orientation, and a software process through a series of model transformations. These allow for traceability of design rationale and accordant decision making, and enable systematic software maintenance. In addition, it opens up the possibility of an incremental software process, in which the software product is evolved over several project iterations.

The Post-Modern Era

From a project point of view, the driving question for success in the post-modern era is directed to the software production schedule: Does the software be used for work? The authority to answer for this question unites all fields of software engineering, and it is embodied in the different perspectives present to one project team. Each team member is equally responsible for contributing knowledge from their field of expertise in order to reach a mutual agreement on the priority of needs and the provision of software services, which should be made available within a defined, short-term time frame. The team consensus on the value of a problem-solving approach to be implemented, i.e. the prospect that it will actually be used for work, is “built up, following practice [. . .] on a case-by-case basis” [159, page 11], rather than following a comprehensive plan. “The details in the plan provide a sense of security in the future and a foundation of knowledge to build from, even if the details are nothing more than a comforting illusion. [Expectation management, which is] closer to reality [. . .] allow a team to adapt as needed.” [86] In this meet-in-the-middle software process, case-based reasoning is essential for the delivery of software. This group decision making is just as team-dependent as it determines the progress of the project. But in focus of discipline in the post-modern era is the timing of the software process, and when the product is in operational use. This era can be characterized by engineering approaches that apply short-cycled, on-demand software development, which aims the continuous delivery of valuable software [30] at a constant pace.

The crisis continues – what’s next?

The software crisis is not passed, and will properly never do. Today, we have arrived in the post-modern era, where “it seems that everything matters a bit yet not much really matters” [134, page 17]. Does history repeats itself? Now, the improvements in software development practice seem to outpace the achievements in discipline. Dijkstra has criticized software engineering for not providing the promised answer to the crisis, and that it will stay that way for as long as “software engineering has accepted as its charter "How to program if you cannot."” [79] Is this criticism justified, when “The other engineering disciplines haven’t found a silver bullet, either.” [134, page 18], and today’s approaches to software development such as visual programming by “No-code allows people who don’t know how to write code to develop the same applications that a software engineer would” [47] realize the pre-modern idea of Application Development without Programmers [150]?

A true software engineering discipline should empower its devotees to “make decisions [throughout the software process] systematically and document them in a useful way” [134, page 18], such that these becomes available to others. The speed focus in postmodern software development is futile, as long as the knowledge about speed improvement practices and their successful application is dependent on a team. Under this condition, achieving accelerated software development remains a one hit wonder. At least it cannot be reproduced by other teams.

In a postmodern software process, the scope of theory is limited to “conditions where that theory is applicable, or is generated by the practice” [159, page 11]. The postmodern project team limits their decision making, on what requirements should be done in the timebox available for the project, to “little narratives” [159, page 11] known as user stories, and ’triangulate’ these “by their different viewpoints of the requirements” [184, page 370]. That is why the requirements given as user stories must be meaningful to most members of the team, not only to the software analyst, in order to serve them as a point of reference for anchoring their experiences, and to allow them to justify their point of view. The team’s ability to anticipate the size of user stories and the scope they can do in a project timebox, determines the progress of the project and sets their pace.

As the “post-modern trait [. . . is] absorbing technology from modernism while putting it to a rather different use” [159, page 13], this dissertation proposes the use of patterns to sustain the decision-making in the planning of postmodern software projects, and empowering project teams to continuously improve their know-how by sharing their experiences and achievements with one another. Patterns have their roots in the modern era and have become commonly known as mean for describing recurring software architectures and designs [46, 90, 91]. But patterns also belong to the post-modern era, as they “are small independent narratives, supported by arguments made on a case-by-case basis in favour of certain [problem-solving] designs” [159, page 18]. They represent a meta-structure for shareable best practices knowledge, which relates the models of the discipline with the modules built by software practices.

Patterns not only help to “bridge any gaps in understanding” [73, page 1343], they also “treat [the problem-solving] design decisions as first class entities” [134, page 18]. These properties of patterns deserves more attention as they are essential to a key feature in the post-modern software process: the project team. “Managing [. . . its decision making by means of patterns] might be the key to end-to-end [requirements] traceability – the solution to capture design rationale, and analyzing the impact of projected changes [on project progress], or harvesting reusable know-how when simply reusing the code isn’t feasible [or intended].” [134, page 18].

There is a way Out of the Crisis for the software engineering discipline that is best described by the words of Max Frisch “A crisis is a productive state. You simply have to get rid of its aftertaste of catastrophe.” Let’s have a try!

A tribute to Michael Jackson

Not the singer. To the software engineer.

Jackson’s approach to “separation of concerns” provides the basis for this dissertation. His point of view in distinguishing problems and solutions, and to make use of problems for the projection of requirements through patterns called *problem frames* are key concepts in the field of problem-oriented software engineering. Along the way, he made important contributions to each of the aforementioned software engineering eras.

During the pre-modern era in 1983, he introduced the Jackson *System Development* method [125]. It takes into account that software development is not just about structured programming. It follows a staged transformation process, which begins with the modeling of the realworld and ends in the implementation of a specification in several steps.

During the modern era in 1995, his joint research with Pamela Zave on the relation of *Software Requirements & Specifications* [126] realizes that software development serves the satisfaction of requirements, but therefore a correctly implemented specification is necessary but not sufficient. Specifications must be derived by taking the environmental conditions of the realword into systematic account. Only then, (when the machine as) the software built upon these specifications (is in operation, it) can establish desired system properties, those as described by the requirements.

During the post-modern era in 2001, Jackson developed the *Problem Frames* approach [128], which addresses the issue that software development by reuse improves the fulfillment of specifications. On the one hand, specifications must represent implementable requirements. On the other hand, specifications must be implementable. Both cases represent recurring situations in software development, for which solutions and know-how exist.

Patterns make these "best practices" that have been found effective in their application reusable and available to others. So, as architecture design patterns serves the description and recognition of recurring solution approaches to build a software, problem frames are patterns applicable in software analysis to classify requirements into recurring problem descriptions for deriving software specifications that represent implementable requirements.

A problem-based decomposition separates requirements into independent problems through projection. When analyzing one problem, the others are considered as solved. What is outside the considerations of the problem frames approach is that the relationship of all projections also needs to be addressed. For example, their dependency on a timely order can be maintained, or their dependency or overlaps on a common solution (design) can be used to put the parts together into a coherent whole.

At this point it should only be mentioned, that using problem frames in requirements analysis establishes a separation of concerns, which is no hierarchical one like in a functional decomposition. It requires separate consideration. Jackson and other fellow frame-artisans have developed approaches to address this need. This dissertation contributes to it either.

A short reading guide

As the postmodern software process also referred to as agile software development in the following favours fast-paced delivery of working software over following a documented software development plan [31], the members in a project team "derive much of their agility by relying on the tacit knowledge [and respective gutfeel estimates of progress] embodied in the team, rather than writing the knowledge down in plans" [35, page 66].

The aim of problem-based project planning is to make this tacit knowledge tangible, and thus reusable in other project and by different teams. Therefore, it integrates patterns from the discipline as a shareable, light-weight documentation of a problem-solving plan and its involved argumentation into agile software development practices.

As table 0.1 shows, on these grounds the research questions

- **RQ 1** How does the project team *determine* speed?
- **RQ 2** How does the project team *adjust* speed?
- **RQ 3** How does the project team *compare* speed?

are investigated and give structure to this dissertation.

The intent of this synergy between agility and patterns is to achieve a win:win situation. To the practice. To the discipline. Both should be able to understand what *speed* needs.

Dissertation	Research Question and Contributions	Models and Patterns *from the discipline to…*	*… sustain* Agile Practices
Part I Motivation, Background, Overview	*introduction to software projects, their planning and performance challenges*		
Part II Problem-Based Project Estimating	**RQ 1** How to *determine* speed?		
Chapter 5 Problem-Based Units of Measure	· Problem-Based FSM Patterns	problem frames	user stories
Chapter 6 Problem-Based Estimating Method	· Frame Counting Agenda	IFPUG functional size measurement	planning poker
Part III Problem-Based Project Adaptation	**RQ 2** How to *adjust* speed?		
Chapter 7 Problem-Based Units of Work	· Transition Templates	transformation schemas, architecture design patterns	software increment
Chapter 8 Problem-Based Adaptation Framework	· One4All View Model	state transition diagrams, UML sequence diagrams	prioritization, amigos, story board
Part IV Problem-Based Project Benchmarking	**RQ 3** How to *compare* speed?		
Chapter 9 Problem-Based Project Baseline and Speed Benchmark	· Requirements Work Packages	measurable problems	work items, time-box, product backlog
	· A S.M.A.R.T. Scrum-Agenda	IFPUG function points, agenda concept	story points, Scrum, sprint backlog
Part V Case Studies Chapter 10 Vacation Rentals Web Application Chapter 11 Student Recruitment Web Portal	*exemplary application and extension of contributions*		
Part VI Conclusion, Future Prospect, Appendices	*pro- and retrospective of the further development of contributions*		

TABLE 0.1 Structure of dissertation, its research questions, contributions, and intension

Contents

Part I.

Software Projects – Perspectives on a Managed Engineering Discipline

Part I introduces the vicious cycle of software projects, which is addressed in this dissertation. One the one hand, there is the management perspective in each software project, responsible for establishing a plan for value delivery. On the other hand, there is the engineering perspective in each software project, responsible for producing a software product, which progresses to plan. Software projects get into trouble, each time the rate of progress (project speed) is not matched with the delivery of value (project size) as planned. Project teams which are in the position to balance these trade-offs are also in control of making their project a success. Chapter 1 Motivation – Empowering software project teams to move faster, presents what speed means to software projects, and why it is challenging to software project processes and teams. It derives research questions to be answered by this dissertation, indicating the gaps to be closed, in order to take advantage of speed control. Chapter 2 Background – On the emergence of product requirements and project teams, discusses the factors that impact project success, and its dependence on plans, which anticipate change. Chapter 3 Research Objective – Sustaining decision making and development by pattern practices, gives insights on the bond of project speed and project size with regard to a defined scope of software product requirements. It proposes the use of pattern practices to sustain both management and engineering tasks in software projects, and which enforce a stabilized project scope that is subject to change control. Chapter 4 Overview – Introducing pre-defined units for planning scope and speed of software projects, summarizes the contributions made available in this dissertation to provide software project teams with a common basis to plan for delivering value and its involved development work both by referring a defined project scope.

1. Motivation – Empowering software project teams to move faster

1.1. The Need for *Speed*

In 1995 Olsen noted, “It is a truism that "software is always late".
This is not only the rueful admission of poor planning, bad process, or poor engineering discipline, but a profound statement of market dynamics in the Information Age: products should be released in the moment they are conceived.” [164, page 28]. He allusioned to the pressure of time-to-market for software products or services as the “primary force behind profit” [164, page 29]. Another 15 years later, this situation has become even more dramatically present in the software business.

As stated by Andreessen (2011) “Software is eating the world.” [16].
“Over the last decades, the value in virtually every industry and very aspect of our lives has shifted from the "atoms" to the "bits" [. . .] software allows for faster innovation” [39, page xxvii] is summarized by Bosch in 2016. And while new technologies and its therewith involved demand for innovation increases, which “are now being adopted by customers *en masse* in a period measuring months [or less], not years” [132, page 33], which King (2010) illustrated by a conclusive graph [132, page 33, and figure 1.4] on how radical these rates for technology mass adoption have changed. Lindegaard (2014) pointed out “the window of opportunity gets smaller [. . . and must be hit] more often in order to create strong return on investments”.

Consequently, Bosch [37, slide 45, published in [22]] demands for “Increasing SPEED [of software delivery, which] trumps ANY other improvement R&D can provide to the company – it is the foundation for everything else. As a process, methods or tools professional, there is only ONE measure that justifies your existence: how have you helped teams move faster?”

Speed-up software projects means according to Bosch (2016) to shorten“the time from identified customer need to the delivered solution in the hands of the customer” [39, page 6], which implies the accelerated responsiveness to emerging user requirements as well as the rapid delivery of the software product, which both make the business value.

This is the field where agile project practices promise to perform best, since they implement a time-boxed software development process of frequent, incremental value delivery, which allows for immediate software adaption to changed user needs. The fast-tracked delivery of “Working software” and “Responding to change” are two key values in agile software development, as documented in the Agile Manifesto [31].

In this context, agile projects are advertised for executing projects in a state of hyper-productivity, which according to Sutherland conforms to “at least the Toyota level of performance, which is four times the industry average.” [204] A commonly perceived, but often neglected issue involved with this promise of reaching accelerated software production due to agile project practice is, that “We know they exist, [. . .] the increases in productivity we see through the course of an Agile project [. . . ,] but we lack any defined way to measure them.” [105]

The challenge of measuring productivity is not specific to agile software delivery. It is a general concern to each project management and organization, which implements data-driven decision making for planning, monitoring, and controlling projects.

In particular, performance data are essential for quantifying supposed project speed (improvements), but these become unattainable, “If you do not intend to gather data carefully enough to be useful for later analysis [in that case] you should not bother to gather it at all.” [114, page 50]. However, “leaders often struggle is in setting the right pace for their digital journey: fast enough to build the business of the future, but not so furious that they lose control.” [24] So, suitable performance data are urgently needed to pass this challenge. In this context, Ebert and Dumke (2007) noted that, we must overcome “the difficulty in IT services and software development [, which] is still that productivity is not measured consistently – if at all.” [83, page 441], which is a prerequisite for providing evidence, and for taking proper action, such that the project team is moving fast(er).

Humphrey (1995) complemented to the use of data-driven decision-making in software projects, that “Once you know your performance rate, you can plan your work more accurately, make commitments more realistically, and meet those commitments more consistently.” [114, page 12] Agile project practice provides an answer to this by “The term "velocity" [which] is, in ordinary language, just a synonym for speed. [...It] denotes the *number of story points achieved in the current project iteration*. Velocity thus defined is a measure of work accomplished .” [156, page 124] as defined by Meyer (2014), who emphasized in addition, that “ Once the user stories [aka software product requirements] have been given individual [...story points] and an iteration starts, the same measure can serve to assess progress. This is where *velocity* becomes useful.
This notion addresses a crucial need which, surprisingly, has been often ignored in pre-agile software development: to provide a clear, measurable, continuous estimate of the speed at which the project is progressing.” [156, page 123]

Unfortunately, story points are “an arbitrary measure that varies wildly between teams. There’s no credible means of translating it into a normalised figure that can be used for meaningful comparison.” [158], and “since there is no standard way of gathering numbers [...] comparing them becomes a futile excercise.” [105]. Ambler (2016) commented on this trouble, “It generally isn’t possible to use velocity as a measure of productivity. [Due to the fact, that] You can’t compare the velocity of the two teams because they’re measuring in different units.” [13], which Alleman (2014) resumed by proposing to establish the “assessment of progress to plan based on pre-defined units of measure. This avoids the opinion of progress” [8], and thus team-unique performance data. Establishing proper units of measure, which are reusable in different projects and teams, would enable comparison of their productivity and provide data for analyzing speed trade-offs.

In summary, “to make [performance] improvement the team should know their velocity.” [204, page 144] One the one hand “to see if a state of hyper-productivity has been reached [and on the other hand, the team should know...] Are we spending effort in a manner which is sustainable?” [105]. Agile software projects follow an evolutionary approach to software development. Consequently, “"It ain’t just about being fast." Sacrificing quality for speed will only slow you down later. Choose a development toolset that helps produce excellent code in record time.” [176] Accordingly, fast-tracked quality software development is only possible by an intertwined discipline for project management and software engineering, which incorporates best practices. Then, “instead of trying to achieve the measurement part [solely], teams [...can] focus on delivering useful, working software” [105], which is a key to value-driven software delivery, by “look[ing] for solutions that rev up productivity.” [176] This paves the way for empowering project teams to move faster.

Or as Putnam and Myers (1992) explained “there is evidence that software reliability (or number of errors) is related to productivity. When productivity improves, errors seem to decline, or, as others put it, when more emphasis is put on quality, productivity increases.” [178, page 2]

It has only been twenty years since, Coplien et al. (1998) have formulated another truism, describing best what this dissertation is about: “Software engineers must produce systems rapidly, but also be sure they are carefully engineered. This is the software engineer’s dilemma.” [64, p. 45]

1.2. Research Questions

The previous section 1.1 motivates *the need for speed* to the delivery of software that adds value, and gives reason for investigating the following main research questions **RQ 1** to **RQ 3**:

RQ 1 How to *determine* speed of software project teams that is comparable?

Research question **RQ 1** is refined into the following research questions **RQ 1.a** and **RQ 1.b**, which are going to be answered by the research work contributed through this dissertation.

RQ 1.a How to establish pre-defined units of scope?

"Before you can estimate job size, you need a consistent and repeatable way to describe a product's size ." [114, page 69] That is, before the team can decide on items to be included in the project backlog, it requires a common understanding [59] about those product(ion) units. Otherwise, they suffer from a bootstrapping problem [56] each time they set up a project plan: what is the baseline they will compare new items to, which accounts for keeping their project planning consistent.

In case the project team refers items back to a recognizable scope of product requirements, planning and involved estimating "becomes a matter of looking at the previously estimated items and finding something requiring a similar amount of work." [58] It is the team approach to reproducibly pack software product requirements into product(ion) units, applicable as backlog items, which drives the predictability of their project plans.

It is common practice in agile projects to make use of user stories as backlog items. These are structured requirements statements that "typically follow a simple template " [60], but whose story sizes [54, page 6] are not "within one order of magnitude" [55, page 53]. That is, the amount of functionality covered by this statement is of variable level of detail, which needs special consideration within the project planning process. User stories do not support scoping or sizing of requirements per se.

It depends on the teams' shared knowledge about particular user stories, usually established by discussion on these, to reach consensus on what they perceive as a comparable scope and respective size of requirements. Pre-defined units for establishing defined sets of software requirements, meaningful to most decision makers in the project, would allow for reuse of their common understanding in different teams and projects. These product requirements' units of pre-defined scope would assist teams in stabilizing the comparability of their project plans and deliverables, providing them with a common point of reference to "bridge any gaps in understanding" [73, page 1343].

RQ 1.b How to estimate scope size?

"The principle reason you estimate the size of a software product is to assist you in planning its development. The quality of software development plans, in turn, generally depends on the quality of the size estimates." [114, page 141] In agile projects, a gamified decision-making process known as Planning Poker [54, chapter 6, page 56] serves the project team to establish size estimates for chunks of software product requirements by reaching consensus on point values for user stories. It is a modern adaption of the Wideband-Delphi estimation technique [34, page 335] used for software size estimation since the 1980s.

The key benefits of requirements estimating in points are (i.) that they make "the unit of estimation abstract, which makes it easier to commit to, and easier to adjust your commitments to" [221, page 161], and (ii.) their self-correcting nature [221, page 160]. It "does not matter if our estimates are correct [...]as long as we are consistent with our estimates, measuring [these point values ...] will allow us to hone in on a reliable schedule." [55, page 63] and "to meet expectations more consistently" [221, page 161].

"Studies have shown that we are best at estimating things that fall into one order of magnitude" [55, page 52]. Consistent requirement estimates demand the identifiability of comparable requirements that require "a similar amount of work" [56], and to join these into groups that are meaningful for estimating [73, page 1343] to most members of the project team. Software project teams that rely on the Planning Game are even in the presence of these recognizable units of software product requirements scope, at risk to an anchoring problem: since the meaning of points assigned to requirements or user stories relates to an estimator's "experience and good feel rather than on formal criteria" [73, page 1343]. "No standard for story points exists. Each development team is free to define a story point as they wish." [225, page 40]

Measuring the size [117, 118] of software product requirements [2] instead of guesstimating these, has not been able to prevail in agile project processes so far [211]. It is supposed to be a worthwhile, but likewise futile sweeping effort [55, page xxiii], which indeed offers consistent size estimates, but which also is substitutional to the ease of the estimating method. "Size is an accurate predictor" [114, page 99] and "the need to estimate 'size' [...] has not diminished" [55, page xxiii] even in agile software projects to form the basis for predictable planning of value delivery. An aspired size measurement method should work on recognizable units of software requirements, such that related measuring results always refer to a defined product scope.

RQ 2 How do software project teams get at speed? How do they *adjust* it?

Research question **RQ 2** is refined into the following research questions **RQ 2.a** and **RQ 2.b**, which are going to be answered by the research work contributed through this dissertation.

RQ 2.a How to establish pre-defined units of work?

There isn't just one way of doing it, but to speed up development, you should "maximize the reusable components so you have less to develop" [164, page 33]. Having a "feature that appears in multiple contexts" [226, page 356] offers "the greatest opportunities for reusing not just certain requirements, but also their downstream work product, including architectural components, design elements, code, and tests. This is the most powerful form of reuse, but we don't often detect the opportunity to take advantage of it" [226, page 357]. Hence, these features or groups of unique software functionality, which represent candidates for reuse, are present in different forms in the software product and production assets available.

"Reusing [...something] that someone else has produced does take a good deal of work. [...But,] reuse is the only currently available technology that shows promise of order of magnitude improvements in software development quality and productivity" [114, page 85]. The challenge is to build and link features in a way, which enables their recognition and instant reuse. The aim is to avoid doing things twice.

Cohesive components, such as "COTS (commercial off-the-shelf products) software packages [...] used as a self-contained solution to a problem [...] to satisfy user needs" [226, page 598], or families of functional commonalities, such as applied in software product line development [226, page 356], have in common that these refer a defined scope of recurring forms of software functionality.

Units of work defined on the level of a process asset (as template for software development, which is "a pattern to be used as guide for producing a work product." [226, page 531]), that makes allowance for this defined scope, “will help the team members perform processes consistently and efficiently” [226, page 530], and provision for alternative candidates by making dynamic linkage of ready-to-use software asset possible.

RQ 2.b How to plan worthwhile work volume?

“First Things First [. . .] The key is to figure out how to deliver the most value the most quickly. [. . .] you want delivers those 20 percent of features that hold 80 percent of the value” [204, page 188]. Agile project practices make use of a prioritized list of software product requirements or user stories as items of a product backlog, which become allocated to the project plan. Features on top of the product backlog have the highest priority to the customer (or user of the software), who decides on these and their order on the list. So, planning the work for the delivery of value is always directed towards user satisfaction and priority as given by this list [226, page 597]. The team picks items from the backlog and creates work packages for their delivery. If more detail regarding the development of such an item is available, the better planning of work volume for a project timebox becomes possible.

For instance, taking in addition the software life cycle into account, brings more insights on the importance of work items for satisfying backbone features of the application. This allows for respective classification of these into need- and nice-to-have ones, which sustains decision making on the project plan. Furthermore, knowledge about the time needed to complete comparable (sizes of) work items provides also for justifying the work volume of the project plan. That is, knowing the speed at which a project team demonstrated successful value delivery is cruicial.

Making systematic use of traceability between desires and deliverables adds to product-based, output-driven planning in software projects, which enables to replay or revise respective decision making and development plans as needed for a doable work volume.

RQ 3 How to *compare* speed of software project teams?

Research question **RQ 3** is refined into the following research questions **RQ 3.a** to **RQ 3.b**, which are going to be answered by the research work contributed through this dissertation.

RQ 3.a How to baseline project plans?

Within an agile project process framework such as Scrum [196], the project plan is represented by the Sprint Backlog, it is referred to as Project Backlog or Project Baseline in the following. It is an approved proposal [19, page 307] for achieving a project product plus a plan of work to be done for its delivery “in order to gain acceptance” [19, page 310].

The project backlog, which reflects the scope of a project plan (project scope), is defined by its set of backlog items. These are work packages [19, page 314] planned for the upcoming (Sprint) timebox to fulfill a particular extent of software product requirements. When "done", these units of work contribute to the project output [19, page 307] (Increment), producing the desired project product [193, page 16] scope.

Planning reasonable work volume for a timebox requires knowledge of project teams’ past performance, which is based on the point values a team has assigned to each work package. These points for each unit of work depends on the unit of measure used by the team for estimating desired product scope, or rather they have used for determining respective software requirements’ size. Accordingly, work packages, done within the timebox coincide with satisfying desired product scope. That is, respectively scored points acknowledge the project baseline and its involved team speed measurement.

In order to baseline project progress to plan on pre-defined units of measure for project team speed, these must be meaningful to both software production (work packages) and the software product (requirements), to leverage commonalities within their scope for project planning. Software project teams take advantage by building on these commonalities for determining scope size and for making the product and production view intertwine in a reusable way.

RQ 3.b How to benchmark the progress of projects?

In agile project practice, working software appraised "done" by the team is the primary measure of project progress [31]. Speed is “The current rate of progress” [21, page 129]. It is “the volume of work accomplished in a specified period of time by a given team.” [103]

“Velocity in agile methods is a measure of the speed the team is delivering at, usually” [221, page 338] determined by “an arbitrary point system that is unique to a given team” [13]. “Ultimately, a team can work out how many points they can do in a timebox and can forecast future work throughput, although this does need a degree of stability in the working environment so that like is being compared with like.” [21, page 106]

“But you can’t really use Velocity to compare” [221, page 170] ”two teams because they’re measuring in different units” [13], used for planning the project: the work to be done, the point values to express the involved volume of work, and the timebox available. It relies upon the teams shared method and experience to make the approved project plan a project performance measurement baseline [124, page 82], which serves them as “reference level” [19, page 303] for assessing “actual progress against expectations” [19, page 309], and for allowing them to measure integrated performance in terms of speed consistently.

Sustaining project teams in the “assessment of progress to plan based on pre-defined units of measure” [8], would be of cross-team and project benefit. It would have the advantage of mitigating the risk of manipulated point values due to error or forced inflation [21, page 251], affecting the comparability of project team speed. Pre-defined units for establishing the software project plan would provide project teams with a common point of reference for objective measurement and comparison of their speed. These need to be made available through the software process in order for them to take effect for the team.

2. Background – On the emergence of product requirements and project teams

2.1. The Software Project Triad

Any “project is a temporary endeavor undertaken to create a unique product” [124, page 5]. Project management is concerned with “planning, [...] monitoring and control of all aspects of the project” [19, page 4 and 309] to achieve the desired product within the expected project constraints. Figure 2.1 shows the competing project constraints of time, scope, and resources affecting one another [124, pages 6 and 7]. Software project management is concerned with balancing these constraints for establishing a sound project plan, one which ultimately defines project success, since project delieverables are compared to it [124, page 99].

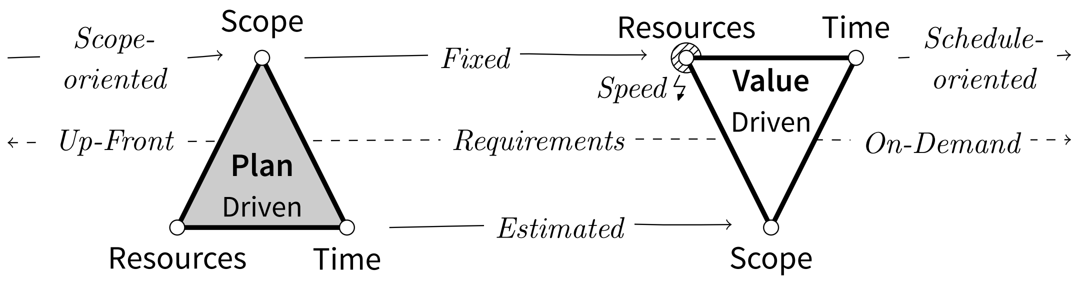


FIGURE 2.1 The software project triad on its head, adapted from [144, page 68, figure 6-1]

On its left, figure 2.1 shows the so-called "iron triangle of project management" as been seen in traditional, plan-driven project management processes. Scope-oriented, waterfall-like managing of software projects means to subordinate the planning of time and resources to a rigor, up-front fixing of the project scope, which is represented by a plan of the “work that is needed to be accomplished to deliver a [software] product [...]” [124, page 103] and “the extent of [its] requirements” [19, page 312]. On its right, figure 2.1 shows the shift of the same competing project constraints, as been seen in recent, value-driven software project management practices. Schedule-oriented, agile project practices means to plan software product requirements on-demand for the delivery of value on time, which is the top-priority. The project team "embraces change" by estimating how much scope they can do within a fix project timebox. Therefore, they need to know their speed.

As reported by The Standish Group [208], insufficient requirements belong to the top three impairments of successful software projects, besides the lack of user involvement and the unability to cope with change. Scope- and schedule-oriented software project management following different emphasis in planning projects for mitigating this risk.

Boehm [35, page 67] argues, that plan-driven methods have their home-ground in high-assurance software, “investing in lifecycle architecture and plans [...] to facilitate external-expert review” [35, page 66], particularly with regard to any shortcomings in the requirements. In addition, this kind of documentation allows to coordinate and to scale for projects in complex settings. Scope-oriented project management works best, when requirements can be determined in advance und remain relatively stable [35, page 66], since “rapid change will make the plans obsolete or very expensive to keep up to date” [35, page 66].

Value-driven methods focus more on “the planning process than the resulting documentation, so these methods often appear less plan oriented as they are” [35, page 64]. They ”derive much of their agility by relying on the tacit knowledge embodied in the team, rather than writing the knowledge down in plans [. . . , which works fine, when their common understanding of the requirements is] sufficient for the application’s life-cycle needs” [35, page 66]. This makes agile project practices preferable for small projects with equally small-sized, stable teams that can handle respective face-to-face communication. Schedule-oriented approaches work best when “requirements are emergent rather than prespecificable [. . . , but] these methods risk tacit knowledge shortfalls, which the plan-driven methods reduce via documentation and [. . .] review.” [35, page 66]

Boehm [35, page 69] concludes, “Although information technology trends are moving us closer to agile method’s emergent requirements and rapid change [. . .], increasing dependability concerns call for measures best implemented with plan-based solutions. To meet these disparate needs, organizations must carefully evolve towards the best balance of agile and plan-driven methods that fit their situation.” They “don’t need just rapid value or high assurance – they need both.” [35, page 67]

2.2. The Four Dimensions of Software Project Speed

According to McConnell [151, pages 3, 4] “Development speed depends on the choice of [effective] schedule-oriented practices [. . .] that enable you to develop faster.” Figure 2.2 shows that software projects “operate along four important dimensions: people, process, product, and [platform, . . .] which can be leveraged by practices for maximum development speed.” [151, page 11]

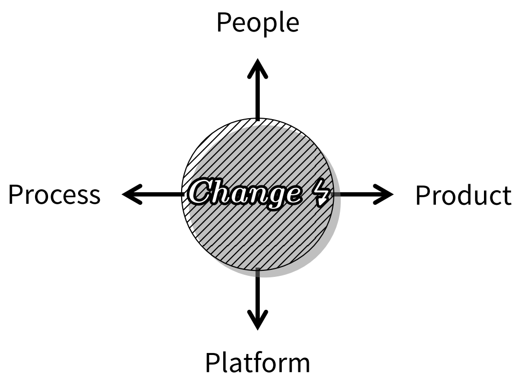


FIGURE 2.2 The four dimensions of software project speed, adapted from [151, p. 11, fig. 2-3]

McConnell [151, pages 3, 4] clarifies also that “When you put effective schedule-oriented practices together with a plan for using them, you’ll find that the whole package provides for dramatic, real improvements in development speed.” [151, pages 3, 4]

This plan for efficient software development is only made possible by engineering discipline “to improve overall development capabilities” [151, page 20] of individuals, teams and organizations, which are involved in software projects. It manifests in the software project management strategy or respective development lifecycle process, and should aspire to balance plan-driven anticipation and value-driven adaptability according to Highsmith [111, page 214] for improving the perpetuity of software projects responsiveness to change. This “is as necessary as it has ever been, especially as software system size and complexity grow” as concluded by Boehm and Turner [36, page 23].

McConnell [151, pages 19, 20] argues further that “For many projects, [this plan] represents a sensible optimization of [the project constraints, given by resources, time, and scope]. [...] Another reason to focus on [software engineering discipline] first is that for most organizations the paths to efficient development and shorter schedules are the same. [...] they just need to get organized! [...] most [software projects] would benefit from setting a course of efficient development first.”

In the following, a brief description of each dimension of software project speed is given, together with a short explanation of their role and their dues for operating software projects.

Product is the “most tangible dimension [...], and a focus on product size and product characteristics presents enormous opportunities for schedule reduction[, ...] limited only by your customer’s product concept and your team’s creativity. ” [151, page 17]

“Because the effort required to build software increases disproportionately faster than the size of the software, a reduction in size will improve development speed disproportionately. [...] don’t shackle your developers by insisting on too many priorities at once. ” [151, page 17] “The product [should be defined ...] in a way that stymies the best effort of the people who are building it.” [151, page 11]

People perform “quickly, or they perform slow.” [151, page 11] “The way people are organized has a great effect on how they can work. [Software projects] can benefit from tailoring their teams to match project size” [151, page 13].

We “know with certainty that peopleware issues have more impact on software productivity and software quality than any other factor. [...] study after study have found that the productivity of individual programmers with similar levels of experience does indeed vary by a factor of at least 10 to 1. [...] Studies have also found that variations in the performance of entire teams on the order of 3, 4, or 5 to 1.” [151, page 12]

The “most effective practices are those that leverage the human potential [...]” [151, page 12] Any “organization that’s seriously about improving productivity should look first to the peopleware issues of motivation, teamwork, staff selection, and training.” [151, page 13] “A signification consideration here is the unavoidable statistic that 49.9999 percent of the world’s software developers are below average” [35, page 65]. Olsen [164, page 32] adds that “On the long run, you get more bang for your buck when you hire top notch programmers.” Chose practices that enable the software project team to excel.

Process “as it applies to software development, includes both management and technical methodologies. [...] Process represents an area of high leverage in improving your development speed – almost as much as people. [...] Organizations that have explicitly focused on improving their development processes have, over several years, cut their time-to-market by about one-half and reduced cost and defects by factors of 3 to 10” [151, page 14].

“One of the most straightforward ways to save time on a software project is to orient your process so that you avoid doing things twice.” [151, page 15] “Michael Jackson (the singer, not the computer scientist) sang that "One bad apple don’t spoil the whole bunch, baby". That might be true for apples, but it isn’t true for software. [...] To slip into slow development, all you need to do is make one really big mistake; to achieve rapid development you need to avoid making *any* big mistake.” [151, pages 37, 39] “Do it right the first time. When you make a mistake, fix it right away [...] Fixing it later can take you more than twenty times longer than if you fix it now.” [204, page 109] “The lessons learned from 20 years of hard knocks can help your project to proceed smoothly. [...]

Half of the challenge is to avoid disaster, and that is an area in which standard software-engineering principles excel." [151, page 15]

"The process leverages people's time, or it throws up one stumble block after another." [151, page 11] "targeting resources [in terms of people in the project team [124, page 446] and reusable software assets] effectively [. . . to] get as much work done each day as possible [. . . and therewith] contribute to the overall productivity." [151, page 16]

Platform "Technology assists the development effort, or it thwarts developers' best attempts" [151, page 11]. "Change from less effective tools to more effective tools can also be a fast way [. . .] Change from low-level languages [. . .] to high-level languages [. . .] was one of the most influencial changes in software-development history. [. . .] move towards componentware [. . .] might eventually produce similar dramatic results." [151, page 17] "The system should be first made to run." [42, page 18]

Each dimension of software project speed provides its own practices to improve the performance of software projects. The challenge is to keep an eye on their fruitful synergy in the presence of change, which in the following is attributed back to a matter of (choosing) proper practices for project scope control, in regard to the specification and satisfaction of software requirements.

New or modified product requirements must be clearly defined, and their size and dependencies need to be determinable, and scalable in a reproducible way. This is a prerequisite to proper scope control as needed for iterative, short-cycled projects, which intend to take advantage of incremental software development. It also makes independent work on respectively synchronized deliverables possible, such that more people can get more work done, which contributes to the overall project pace.

User involvement in this connection, not only means to proactively address the customer's ignorance of purpose [42] and priority [227]. It also calls for investing in preparatory training of each team member to fast-track their orientation on how the product requirements are (in general) turned into desired deliverables. Awareness of worst practices and knowledge about best ones contributes to the raise of quality and the avoidance of disasters, which ultimately pushes a project's (team) performance. People must be enabled to adapt their actions quickly and thus become responsive to changed project conditions.

Establishing bidirectional traceability of requirements from their emergence to their implementation is a key to this challenge. This is where software architecture becomes vital for the decision making and development as utilized within the project process.

Architecture holds required traceability links, which ease reuse and integration, by describing (a defined scope of) commonalities between desired product functionality and present technology (platform). By appropriate application, these (comparably scoped) commonalities are a starting point for targeting the resources available (in each dimension) effectively.

Leveraging these commonalities by means of patterns provide teams with a baseline for their benchmarking and for exploring (alternative) options of action. They can compare to the performances and benefit from experiences, which have been made in problem-solving cases of other projects. It facilitates doing it right the first time, which increases the overall project quality, and consequently (and especially in incremental development) speed-up (team) performance.

2.3. The Fuzzy Front End of Software Projects

Since each software project is an unique endeavor, each step towards delivery of the project product is accompanied with uncertainty, indicated by the lack or variability of information available for proper decision making. To fast-track software projects, these require to start as soon as possible and to proceed as planned, which is hindered, when project options cannot be fixed sufficiently, to turn them into action for the project to proceed.

There is the risk of wasted time during the fuzzy front end of software projects, which "is the time before the project starts" according to McConnell [151, page 44]. This work considers the entire software project planning horizon from this fuzzy front end, up to the point where software development starts. This horizon ends in plan-driven as well as in value-driven software delivery at the time point, where the requirements are approved complete, and the project scope is documented in the project plan.

In this context, two forms of variability or fuzziness that yield uncertainty must be controlled or properly responded: change in early project stages affecting its timely initialization, and change that comes later during the project, affecting formerly estimated conditions and the project progression as planned. In the following, change with regard to product scope and the people dimensions in figure 2.2 is further detailed, since these have the greatest impact on speed, as motivated in section 2.2.

The need to anchor the project scope Figure 2.3 shows the so-called Cone of Uncertainty given by a bold, tapered graph. It shows " the best-case accuracy that is possible in software estimates [with regard to the project scope] at different points in a project " [152, page 37]. Both plan-driven as well as value-driven software projects have to execute the steps annotated along the horizontal axis of the graph from *initial concept* to final *software product*, even though in different detail and number of repetitions. "The vertical axis contains the degree of error that has been found in estimates created by skilled estimators at various points in the project [...] estimates created very early in the project are subject to a high degree of error [...where] The total range from high estimate to low estimate is 4x divided by 0.25x, or 16x!" [152, page 36].

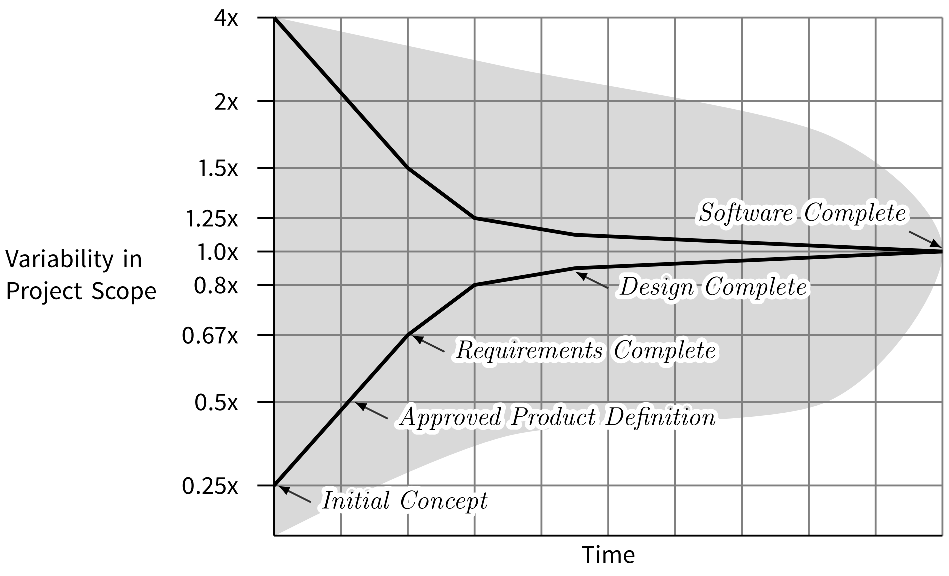


FIGURE 2.3 The Cone of Uncertainty, adapted from [152, p. 37, fig. 4-2, and p. 38, fig. 4-3]

Figure 2.3 shows by the gray shaded area, "what happens when the project doesn't focus on reducing variability – the uncertainty isn't a Cone, but rather a Cloud that persists to the end of the project. [...] You must force the Cone to narrow by [making decisions about what the project will deliver or not that remove] sources of variability from your project." [152, page 38] The Cloud of Uncertainty indicates, that change which comes late to the project, i.e. after the project plan is fixed, expands the agreed project scope, if not controlled or responded properly. In this regard, the Cloud of Uncertainty marks deviation of the project as planned, which in contrast is represented by the graph for the Cone of Uncertainty.

Laranjeira [136, page 517, figure 5] has found that the accuracy of the software estimate (consideres as relative size range) and its therewith involved software project planning depends on the level of refinement of the software product (requirements). "The reason the estimates contain variability is that the software project itself contains variability. The only way to reduce the variability in the estimates is to reduce the variability in the project." [152, page 36]

In contrast, remaining too long in the fuzzy front end, would not provide for more accuracy. As figure 2.4 shows by the time-accuracy curve, more time spent on requirements specification would have a reversal effect on their accuracy after a certain point in time. Value-driven project practices accept, that the software product requirements cannot be specified in all detail in advance, taking into account that users are rarely aware of what their needs really are [42, page 17]. Agile requirements cope with the fact that software product requirements can only be approximations of the users needs, so that a bit of vagueness remains, which is seen as chance to do better rather than risk to fail. Plan-driven project practices stop requirements specification and thus the approved project baseline at a later point in time, where the risk of deprecated requirements and their possible end-up in analysis paralysis becomes much closer. That is, plan-driven projects start with a wider cone and respective project scope size than value-driven ones, which makes them more sensitive to change.

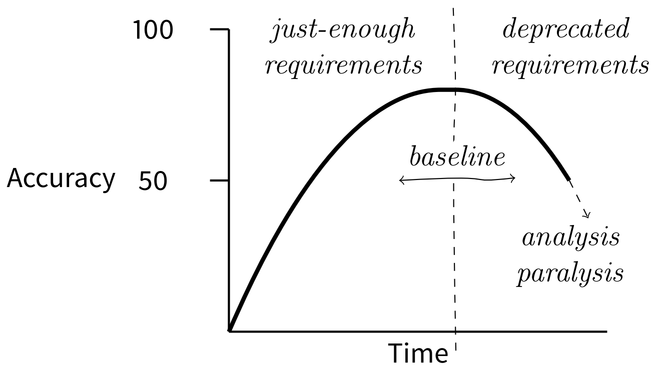


FIGURE 2.4 Time-Accuracy Curve, adapted from [55, page 50, figure 6.1]

The need to bootstrap the project team relates to Brook's Law of *Adding manpower to a late project makes it later*, which Brooks [43, page 23ff, chapter 2] illustrated by the fact that it takes time to people to ramp up. Newly formed teams or assigned members require extra-time for coordination, training, and communication on the work to be done, which slows down the overall performance of the team, and thus makes an already late project later, when not accounted for wisely.

This is also supported by Tuckman [213]'s 4-stages model of group development, where each newly formed team develops sequentially through the stages of

1. **Forming**: team members getting to know each other.
2. **Storming**: team members assert their differences, which need to be aligned.
3. **Norming**: team members become constructive and learn how to work effectively as a team.
4. **Performing**: team members performing to their best.

Teams in stage four have the best collaborative maturity. Their members share lessons learned, resulting from experiences made and assumptions built on common projects, they estalished means for talking the same language to each other, which finally make performance improvements become effective. It can be supposed that consensus reached in a stable team is of high reliability [146]. Stage four is only accessible by means of team stabilization, either regarding team membership, or regarding the conceptual integrity [42, page 11] of a project. It is a prerequisite to speed up development, and also an indicator for effective maintainance of a teams' shared understanding about how to and what product to build.

It cannot be ignored that project staffing is volatile even in agile projects [179]. Notwithstanding, especially agile projects are "Assuming that there is stability and constancy in the team" [21, page 129], i.e. a stable team [220], for doing justice to the creed to "trust the team", since they are "relying on the tacit knowledge embodied in the team, rather than writing the knowledge down in plans" [35, page 66]. This makes them more sensitive to personnel turnover than plan-driven project practices.

Bootstrapping teams means to fast-track their initialization (stage 1. to 3.) when projects start, and particularily in the presence of change, by empowering their collaboration and knowledge sharing.

3. Research Objective – Sustaining decision making and development by pattern practices

3.1. Balancing Project Trade-Offs

As motivated in section 2.1 The Software Project Triad, software project management is about balancing the constraints of scope, time, and resources for establishing a project plan, and controlling project progression according to it. Figure 3.1 continues discussion on these project constraints for schedule-oriented project management practices, as has been started by figure 2.1 on page 8.

The left hand triangle in figure 3.1 as suggested by Leffingwell [144, page 68, figure 6-1], presumes not only a $Fixed$ project **Time** box, but also a $Fixed$ commitment to the (speed of the) **Resources** available (at which the project will proceed). This coincides with one of the principles behind the Agile Manifesto [31], which is that the team "should be able to maintain a constant pace [aka speed] indefinitely" [30]. That way, delivery of desired project **Scope** is to be $Estimated$ by its bound to a $Fixed$ project deadline and pace, and thus can only change accordingly.

The respective estimation equation is "Velocity [of **Resources**] x **Time** = Delivery [of **Scope**]. Once you know how fast you're going, you'll know how soon you'll get there" as proposed by Sutherland [204, page 144]. In other words, once you know your speed, you know how much value you can reasonably plan for delivery by the project, given as a "quantified measure of the project's scope" [33].

Consequently, "the relevant estimation parameter is [size, e.g.] how much functionality of given quality will fit into that time box." [225, page 37], which applies to the speed of the resources (how much functionality is doable?) as well as to the project scope (how much functionality is desired?). So, "A trade-off has two sides. [...] Changes in one corner always impact at least one other corner." [33] This is where the left hand triangle of schedule-oriented project management is going round in circles. It conflicts with another agile principle, namely to "Welcome changing requirements, even late in development." [30], by impeding to trade off change to project scope against one of the others project constraints, which can only result in cutting off value from delivery, to meet $Fixed$ resources and time, when following the left hand triangle.

That way, this dissertation proposes the right hand triangle in figure 3.1, which gives a more realistic illustration on schedule-oriented software project management. The only thing that can be reasonably presumed is the $Fixed$ project **Time** box or **Schedule**d project deadline.

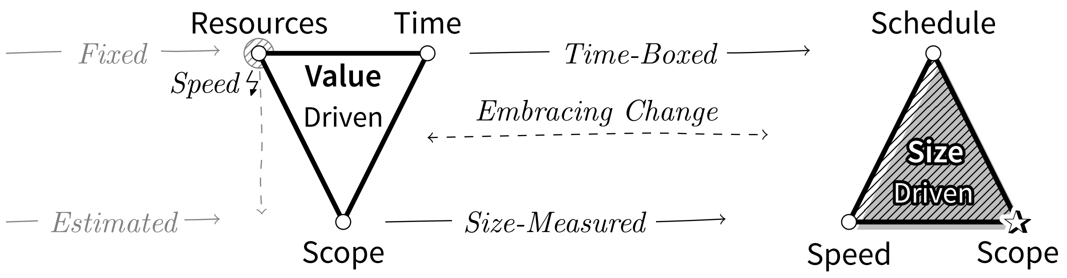


FIGURE 3.1 The project trade-off triangle, cf. [151, page 126, chapter 6.6, figure 6-10]

In order to fix the project schedule for value delivery, or as Wiegers noted “To estimate the [time] effort needed to complete a body of work, [...] you need some measure of requirements size [as involved with the project scope], along with knowledge about the development team’s productivity [as a measure for the project team speed].” [225, page 37], for which he gives an estimating equation comparable to that of Sutherland, see page 15. In the following, it is here adapted from Wiegers [225, page 37] and given as subscripts for taking the project constraints into account: $\text{Resource}_{Productivity} \times \text{Time}_{Effort} = \text{Scope}_{Size}$, or alternatively as

$$\underbrace{\text{Time}_{Effort}}_{\textbf{schedule}} = \underbrace{\text{Scope}_{Size}}_{\textbf{scope}} \div \underbrace{\text{Resource}_{Productivity}}_{\textbf{speed}}$$

for highlighting the relation of this equation to the right hand triangle in figure 3.1. As can be seen, the **schedule** as one of the project constraints represented in this fundamental estimation equation as “[Time] Effort is a function of size.” according to Pfleeger et al. [175], regarding the project scope and speed.

Resource “$Productivity$ [or **speed**] indicates what quantity of functionality [, which is representable as $Size$] the team can realistically [...do] in a [...project timebox].” [225, page 37], and “[...] The size of the product, and hence the $Size$ of the project [**scope**], depends on the size of the requirements.” [225, page 37]. So, “Because software $Size$ is usually the most influential factor in determining [the team’s productivity], good estimates of [requirements] size are critical to good [speed] estimation.” [175, page xv], and impact project planning respectively, see figure 3.3 on page 19.

So as to trade off accelerated project progression within $Fixed$ deadline, you have to balance project speed against project scope, which requires a proper quantification of a software’s functionality. In this context, $Size$ data provides a basis for justification, which is especially valuable in situations common to project management where “Arguing [...] doesn’t work, particularly when they are your superior or customer. What does work is using decision-making data. That is the benefit of [quantified] project trade-offs.” [33]

For instance, it is a fact that “Most software customers initially want more than they can afford. [...] that means that they have to bend either their ideas about the product or their ideas about the resources they are willing to commit. ” [151, page 170, figure 8-3] Good $Size$ data represent “quantified trade-offs [..., which] gives executives data to evaluate the alternatives for taking advantage of opportunities and recovering from problems.” [33] So, “If you want to change [or decide on] one of the corners of the triangle, you have to change [or bend] at least one of the others to keep it in balance. ” [151, page 126] “If you have a quantified measure of the project’s scope (business value) and you follow best practices when building the project schedule and [required project speed], you can represent your sponsor with quantified trade-offs between the[se three constraints] of the project plan. This data-based decision making [...] is a good platform for approval.” [33]

3.2. Designing for Change

Section 2.2 The Four Dimensions of Software Project Speed motivates the use of schedule-oriented practices along the dimensions of people, product, platform and process for improving development speed, but as given by figure 2.2 on page 9 each of these is sensitive to change.

Section 2.3 The Fuzzy Front End of Software Projects carries on with discussing the burden of change for software project planning, illustrating that it most frequently becomes manifest in the variability and complexity involved with software product requirements.

In order to limit this burden, and to allow for choosing proper practices and assess their efficacy both in the presence of change, a means to “Change Control […which] is about providing decision makers with the information that will let them make timely and appropriate decisions to modify the planned functionality.” [225, pages 147, 148] must be carefully designed, which is aimed at the software product requirements.

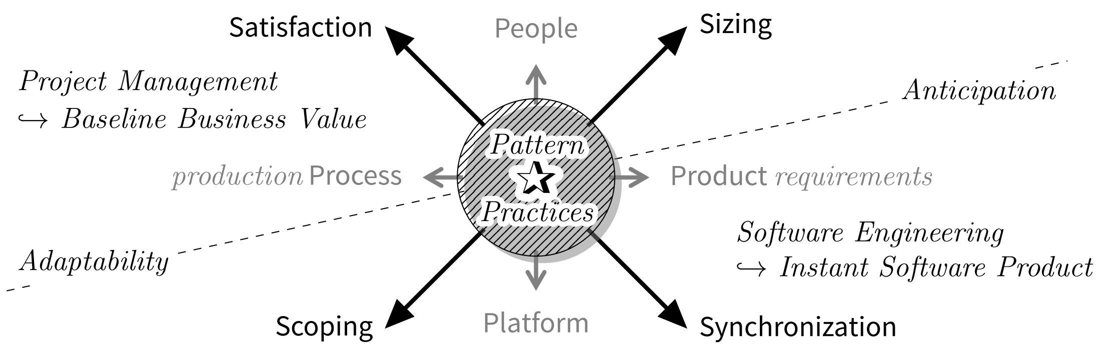


FIGURE 3.2 The four dark spikes of designing the responsiveness to change

Figure 3.2 advances accordingly. It introduces four contributors to data-based decision-making in software project planning respecting the sizing, synchronization, scoping and satisfaction of software product requirements.

Sizing of the software Product provides for a quantification, which allows for determining the number of People required to complete the project on deadline. **Satisfaction** of some requirements is acknowledged by the People making a value-driven project team, and follows the approval procedures as defined by the project Process. Software *project management* relates requirements sizing and their satisfaction for obtaining important insights to *adapt* the *business value* of a project plan.

The relevance of **Synchronization** is twofold: first, it means to make Product *requirements* dependencies tangible. Second, synchronization is also about seamlessly integrating desired software functionality to the technology Platform of intended use. **Scoping** serves the recognition of chunks of requirements and their accordant assets within the used software Platform and the project Process, both as applied for assembling the product. Within the *software engineering* tasks of software projects, requirements synchronization and their scoping determine the capability to *anticipate* accelerated *software production*, by planning for quality, in terms of satisfied user expectations, and reuse of software product(ion) assets.

Consequently, designing responsiveness to change requires means to sustain the stability and separation of requirements. Therefore, patterns are applied to spot software product requirements and to ensure reproducibility of their scoping, sizing, synchronization, and satisfaction. This establishes pre-defined *pattern-based units* applicable in software project planning, which are useful for anchoring change control to planned software functionality and a qualified selection of speed improvement practices.

Table 3.1 exemplifies the relation of speed improvement strategies and their implementation by project management and software engineering means for enabling responsiveness to change. In its left column, table 3.1 gives three speed improvement strategies as suggested by Olsen [164, page 33]. The other columns arrange the parameters usable for controlling change to software product requirements as introduced in figure 3.2, respectively.

Speed Improvement Strategies according to Olsen [164, page 33]	Project Management Techniques. . .	Software Engineering Techniques. . .
	. . . are aimed at ✰ software product requirements, cf. figure 3.2	
Start sooner.	Adaptability to business value	Anticipation of software product(ion)
Reduce work volume.	Satisfaction	Synchronization
Work more quickly.	Sizing	Scoping

TABLE 3.1 Speed improvement through responsiveness to change

Table 3.1 shows how “Good practices tend to support one another.” following McConnell [151, page 18]. Speed improvement strategies become achievable through project management and software engineering techniques synergy.

Balancing value-driven adaptability and plan-driven anticipation of software product requirements as postulated by Highsmith [111, page 214] and already discussed in section 2.2, enables to *start sooner* due to a(n en)light(ened) and weighted project plan. In this context, patterns serve to **spot** requirements in a reusable way, which eases their **classification** into assets that make the business value and that are likewise relevant to software production.

In order to *reduce work volume*, (re)considering the satisfaction and synchronization of planned software functionality provides for coping with involved **complexity**. Limiting the work to be done coincides with requirements prioritization, deciding which are the need- and nice-to-have ones for satisfying user expectations. Requirements depend on each other. **Separating** these into recognizable groups or families, which can be best done on by use of patterns, and making their relation tangible, not only allows for selecting the essential ones to be planned first, but also it helps avoiding duplicated work.

Stabilizing the common understanding of planned functionality in software project teams, lets them *work more quickly*. Sizing and scoping of software product requirements must therefore go hand in hand, for sustaining lessons learned and for providing **continuity** to the teams' view on requirements, since these involve practices and benchmarks applicable to any next project plan.

Designing for change in software project planning is about establishing means that provide decision makers with instant options for action. This is achievable through the use of Pattern Practices for software project planning.

3.3. Planning for Value

“Value is subjective” [95, chapter 4.1], it is in the eye of the beholder. So, which view on value should be reasonably planned with before starting a project? From a business perspective, this value may be the speed of launching new software and the monetary gain expected from it. From a user perspective the value may depend on satisfied expectations and the attractivessness of their realization as experienced by the user. From a development perspective the value involved with producing software may relate to the ease of mastering quality and involved delivery and maintainance concerns. These different perspectives on value melt into objectives for software projects and their respective teams, namely the project plan, which is based on software requirements. These have to be negotiated and finally be formed to a mutual agreement.

In accordance with Doran (1981), for dicussing software functionality, it should represent “meaningful objectives [. . . , which] frame a statement of results to be achieved” [82, page 36], and which “give quantitative support and expression to [. . .] beliefs.” [82, page 35] For externalizing the value adhere to those statements of desired software features, determining their (functional) size provides for quantification as instrumental in the project planning process.

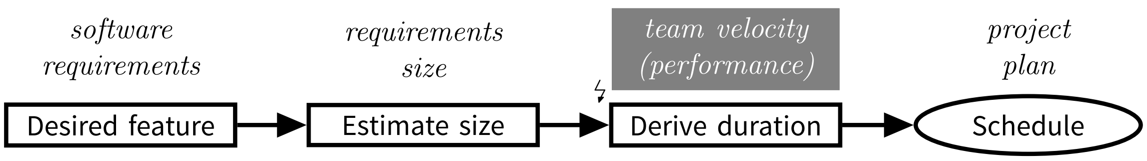


FIGURE 3.3 Estimating provides input to project planning, adapted from [55, p. 34, fig. II.1]

Figure 3.3 shows the use of **Estimating size** for **deriving the duration** of a project, which according to Cohn (2005) is a “key tenet of agile estimating and planning” [55, page 39]. The only catch is in order to derive duration, “we [need to] know the team’s velocity [. . . to] divide [requirements] size by velocity to arrive at an estimated number of iterations. [. . . This derived duration can be turned] into a schedule by mapping it onto a calendar.” [55, pages 38 and 39]. Hence, the relation of planned feature size and doable feature size is crucial to agile project planning and to establish meaningful plans. Otherwise and as made obvious by figure 3.3, it can be expected, that “errors in judgement [. . . of size] will compound themselves throughout the entire [. . . project planning]” [82, page 35] yielding project plans which are hard to fulfill.

On its bottom, figure 3.4 shows the dependence of the three project constraints of scope, speed, and schedule, in order to take advantage of size for indicating the value involved with a project plan. It maps the project planning framework as illustrated in figure 3.3 to the use of size for quantifying value, which is introduced by the right project trade-off triangle in figure 3.1 on page 15.

To continue discussion of figure 3.4, size-driven project planning starts with a particular **scope** of software requirements and a size estimate of these. For example, let’s assume the estimated size of a set of requirements is 42 points. This first project constraint makes use of size for quantifying the value of a desired product based on the scope of its software functionality.

In the following, this requirements estimate determines the **speed** or rate of progression needed for delivering desired software functionality. That is, requirements size becomes a measure for planning the team performance, aka velocity required to complete the project. In this example, the team must be capable to deliver 42 points in a project (iteration). To this second project constraint, size quantifies speed as a value regarding required productivity in a project.

Finally, the two aforementioned project constraints of scope and speed must be balanced to fit into a **schedule** or the time available for the project. The ratio between desired feature size and doable feature size becomes of importance to decide on the planned value, e.g. the size of the project plan. To conclude the example, it may be the case that there are no resources available or teams capable for establishing a project, which delivers 42 points. So, the software requirements must be reconsidered to decide, which of these are of most value and thus of highest priority to be scheduled in the project plan. The third project constraint applies size for quantifying the limits within the prioritization of planned value can reasonably take place, which is output as a **size**d-project plan.

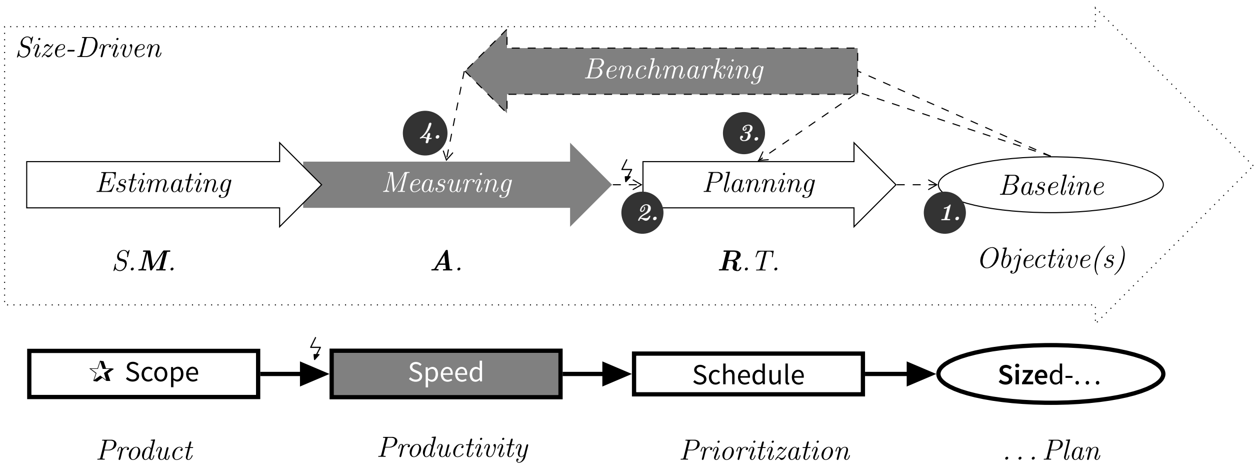


FIGURE 3.4 Size-Driven Project Planning is S.M.A.R.T. (cf. tab. 3.2)

Size-Driven Software Project Planning

On its top, figure 3.4 shows that size-driven project planning must aim at *1.* "Establishing a baseline [...one, which represents] a mutual agreement and expectations among the project [team ...] regarding the product they're going to have when they're done." [225, page 151].
The gray-shaded elements in figure 3.3 and 3.4 highlight the gap to be closed for establishing such "a specified reference point [...by] stable well-defined units for [...software requirements and accordant size-data] that serve as a comparison for [...project plans]" [93].

Therefore, *2.* requirements estimating as applicable for decision making in software projects is developed further to a measurement approach of these. In order to care for reproducibility of size-data regarding their consistency and comprehensibility, defined units for S.M.A.R.T. objectives (cf. table 3.2) are established to guide and thus stabilize the project planning, and to result a product-based project baseline, which is of combined use for management [19, page 307] and engineering [19, page 313].
Due to these S.M.A.R.T. project planning units, a multi-purpose project baseline ready to benchmark becomes available, which *3.* enables the recognition of available resources, and to classify adherent performance data. This not only *4.* helps to set up doable plans, but also provides for planning options. As illustrated by table 3.2 and discussed in the following section on page 21, building these units on the basis of patterns, assists to "Establish more teamwork" [82, page 36], one which can strive for value delivery.

Following Doran [82, page 35], planning means to define "The establishment of [meaningful] objectives and the development of their respective action plans [,which] are the most critical steps in a company's [project] management process."

size-driven	project planning		takes advantage of	pattern-based	units
scope	**S.**	specific	“target[ing] specific area of improvement”	problem functionality	objective
	M.	measurable	“quantify[ing] or at least an indicator of progress”	functional size	
speed	**A.**	achievable	specifying how to do it	solution alternatives	action
schedule	**R.**	realistic	“stat[ing] what result can reasonably be achieved, given available resources”	speed benchmarks	plan
	T.	time-bound	“specifying when the result(s) can be achieved”	project iteration	

TABLE 3.2 S.M.A.R.T. criteria for project planning units, adapted from [82, page 36]

Table 3.2 illustrates, how patterns can be applied for creating desired planning units. For capturing meaningful objectives, patterns used in software requirements analysis can guide the framing of meaningful objectives, which are quantified according to their functional size. For developing respective action plans, patterns known from software architecture and design can become effective. The ones that have demonstrated to match some problem functionality, which is indicated by the availability of respective speed benchmarks that are given as functional size delivered per project iteration, provide for fast-tracked access to potential development assets. In any case, patterns make both revision and replay of the decision making as well as of the development in software projects possible. They are a first-class point-of-reference within diverse software projects and teams for establishing communication and collaboration among them.

Pattern Practices

Patterns represent a ’storage of wisdom’ [94, page 23 and 24] of the good and ugly lessons learned. They encourage learning from the best or worst practices (anti-patterns), and record the knowledge and experiences made owing to recurring problem solving cases in a generic often templated way. The use and maintainance of patterns in software development projects complies to the application of standards. In this regard, “measuring conformance to them and continually trying to improve them [implements a long-term improvement strategy (based on reuse, which)] is necessary if you are to compete well” [94, page 5]. If lessons learned get lost or are hard to access, a (project and its) team is not in the position to accelerate its performance.

Knowledge Sharing

Patterns provide a structured way to capture and share lessons learned [151, page 50]. They introduce a common vocabulary that facilitates communication and mutual understanding. Due to its generic style, a pattern can be applied in many problem solving cases. In this way, patterns capture wisdom, which becomes practicable in manifold domains.

Power Building

Patterns assist not only in the continuous improvement of processes and products, but also in the personal development of people, or in (project) organization's staff development. 'Brain drain' is present at any scale. So, "If your people are not *all* experienced or geniuses, [or respective resources are at risk (to lack required maturity)] You need to store *their* hard-earned wisdom in *your* defined process." [94, page 23 and 24], which is achievable on the basis of patterns. Novice team members benefit from pattern usage taking them as sparring partner for their training. The experienced team members also benefit from patterns which represent the common practices of an (project) organization, to assess and classify their performance. Patterns provide a means to speed-up adaption.

Quality Assurance

Patterns enhance the technical review(ability) [151, page 69] of (project team) work, since their use creates a scaffolding of conceptions, which the team is familiar with, and which they are able to utilize as a starting point or reference for sustaining their project (and team play). Patterns support the identification of reusable resources, but also sources of known errors. In this sense, patterns are a means to reduce the risk of replicated work and the additional cost [151, page 15]. Since, "the lowest defect rate also archieves the shortest schedule" [151, page 69], its worthwhile to build software projects on the basis of patterns. They help to anticipate emerging maintenance effort and implement a strategy to build-in-quality, both which are costly to handle if not considered from the beginning of a project. Patterns of software development make a team live happily in and after projects. They work like a safety net for the project team. Through their use, the team ensures a minimum level of quality that protects the product and thus the project's success from major harm. Ideally, they speed up collaboration in software projects.

4. Overview – Introducing pre-defined units for planning scope and speed of software projects

4.1. Contributions

Table 4.1 organizes the research objective of this dissertation, which is about *empowering software project teams to move faster by sustaining their decision making and development practices on the basis of patterns*, along three strands, each of which constitutes an own part in this work: Problem-Based Project Estimating, Problem-Based Project Adaptation, and Problem-Based Project Benchmarking. It classifies research questions, contributions, and relevant publications co-authored by **Ina Wentzlaff** accordingly, and gives an overview on the contributions **C 01** to **C 06** worked on in this dissertation to respond the research questions involved with **RQ 1** to **RQ 3** in section 1.2.

Part	Research Objectives	Research Questions	Contributions	Publications
II	Project Estimating	**RQ 1** How to *determine* speed?		· Côté et al. [68] · Wentzlaff [223]
		· **RQ 1.a** How to establish pre-defined units of scope?	· **C 01** Problem-Based Functional Size Measurement Patterns	
		· **RQ 1.b** How to estimate scope size?	· **C 02** Problem-Based Functional Size Measurement Method	
III	Project Adaptation	**RQ 2** How to *adjust* speed?		· Schmidt and Wentzlaff [191] · Côté et al. [69] · Côté et al. [70] (best paper award)
		· **RQ 2.a** How to establish pre-defined units of work?	· **C 03** Transition Templates	
		· **RQ 2.b** How to plan worthwhile work volume?	· **C 04** "One4All" View Model on Software Architecture	
IV	Project Benchmarking	**RQ 3** How to *compare* speed?		· Section 13 Future Prospect reports on exemplary results from project practices
		· **RQ 3.a** How to baseline project plans?	· **C 05** Problem-Based Project Baseline	
		· **RQ 3.b** How to benchmark the progress of projects?	· **C 06** Problem-Based Speed Benchmark	

TABLE 4.1 Overview on research objectives and contributions of this dissertation

The following paragraphs describe in brief the scientific contributions **C 01** to **C 06** established in this dissertation to answer the research questions involved with **RQ 1** to **RQ 3**.

C 01 Problem-Based Functional Size Measurement Patterns

Problem-Based Functional Size Measurement Patterns serve to establish Requirement Work Packages, which are equally suitable for classifying and measuring a defined scope of recognizable functional user requirements.

Problem-Based Functional Size Measurement Patterns result from combining problem-oriented software engineering based on Jackson's problem frames [128] with the base functional components measurable within function point analysis [92], which has origin in the work of Albrecht [2]. They make "proxy"-based estimating according to Humphrey [114, page 117, chapter 5] available within early software size measurement for determining the scope of functional user requirements expressed by a function point value in a reproducible way.

Problem-Based Functional Size Measurement Patterns provide software project teams with a shareable point of reference, for anchoring their decision making and adjusting their development activities. This dissertation presents a set of 17 problem frames summarized in table 5.5 on page 72, that consitute pre-defined units for scoping of software product requirements.

Chapter 5 elaborates this contribution **C 01** Problem-Based Functional Size Measurement Patterns, which addresses research question **RQ 1.a** How to establish pre-defined units of scope?

C 02 Problem-Based Functional Size Measurement Method

The Problem-Based Functional Size Measurement Method executes function point analysis on Requirements Work Packages, which are established by **C 01** Problem-Based Functional Size Measurement Patterns. It is an early software size measurement approach for determining function points for a defined scope of software product requirements. It utilizes Requirement Work Packages as units of measure for functional size, which are developed in this dissertation.

The Problem-Based Functional Size Measurement Method introduced in this dissertation, is a requirements sizing method (Frame Counting Agenda) in table 6.2 on page 82, which applies the functional size measurement process of the International Function Point Users Group standard IFPUG [117] to work on **C 01** Problem-Based Functional Size Measurement Patterns.

Each step of the Frame Counting Agenda is equipped with validation conditions given in table 6.3 on page 86 to safeguard its accordance with IFPUG [117], and to ensure that different software project teams are enabled to determine consistent sizes given as function points for comparable Requirement Work Packages.

Chapter 6 elaborates this contribution **C 02** Problem-Based Functional Size Measurement Method, which addresses research question **RQ 1.b** How to estimate scope size?

C 03 Transition Templates

Transition Templates are patterns of patterns, representing those shared configuration properties which are meaningful to intersect patterns of requirements analysis with those of solution design. Transition Templates are built on transformation schemas as known from structured analysis and its use by Ward and Mellor [217, page 41, section 2], specially tailored for establishing problem-based units of work.

Transition Templates are synergies of **C 01** Problem-Based Functional Size Measurement Patterns and commonly known patterns of software architecture design listed in appendix D on page 281 to determine "architectural blueprints" that fit a Requirements Work Package. These provide the software project team with a reasonable (software engineering) plan of anticipated work, needed for accomplishing the desired project product.

This dissertation introduces the concept and use of Transition Templates and provides several examples for their application in the domain of web engineering.

Chapter 7 elaborates this contribution **C 03** Transition Templates, which addresses research question **RQ 2.a** How to establish pre-defined units of work?

C 04 "One4All" View Model on Software Architecture

The "One4All" View Model on Software Architecture paves the way, to make software development problems absorb into the technology platform intended to use. It refines the "4+1" view model on software architecture from Kruchten [133] to work on the basis of patterns exclusively, utilizing **C 03** Transition Templates at its core.

The "One4All" View Model on Software Architecture enables early exploration of engineering options for action in a technology-independent way. It supplies software project teams with instant design solution alternatives for recurring software product requirements on the basis of patterns.

In addition, the "One4All" View Model on Software Architecture accounts for identifying backbone functionality within the software lifecycle, allowing to plan belonging units of work with respective priority in addition to their function points.

Transition Templates make late binding of software development problems to particular solutions available. Within the "One4All" View Model on Software Architecture as proposed by this dissertation, they provision maximum flexibility and just-in-time decision making to software project teams for establishing a prioritized (project management) plan of work to be done next. This is a key to change-tolerant design and to fast-track involved software development.

Chapter 8 elaborates this contribution **C 04** "One4All" View Model on Software Architecture, which addresses research question **RQ 2.b** How to plan worthwhile work volume?

C 05 Problem-Based Project Baseline

The Problem-Based Project Baseline builds on items, which are in equal measures meaningful to both the software product and the software production process. These items form "units", which are established by intertwining **C 01** Problem-Based Functional Size Measurement Patterns, for creating a defined scope of product requirements, and respective **C 03** Transition Templates, for suggesting a plan of work that produces desired project product output.

By means of a Problem-Based Project Baseline which built on requirements as recommended by Wiegers and Beatty [226, page 366, figure 19-1], this dissertation aims at addressing the essential complexity of software projects as discussed by Brooks [42] and Wirth [227]. It takes into account that speed trade-offs within time-boxed software projects, are decided in dependence to the product requirement scope, which is to be satisfactorily "done" by the project team. Therefore, it proposes pattern-based means to rigidly enforce scope control of product requirements in software project management following McConnell [151, page 319, chapter 14], and to fix project size estimates and work plans accordingly.

Making reuse of items as defined by Requirements Work Packages in different project baselines possible, is fundamental to the predictability of value-driven, product-based project planning [19, 21], and its involved performance-based software project management [7, 124]. This is illustrated by applying the Problem-Based Project Baseline to an agile project process framework [193].

Chapter 9 elaborates this contribution **C 05** Problem-Based Project Baseline, which addresses research question **RQ 3.a** How to baseline project plans?

C 06 Problem-Based Speed Benchmark

The Problem-Based Speed Benchmark indicates the rate a software project team is delivering value within the project timebox available. It can be determined by application of A S.M.A.R.T. Scrum-A·Gen^{E}DA as is introduced in this dissertation. It enables benchmarking of the software process by producing size-driven speed benchmark [48] that represent the pace at which a software project team satisfies user expectations. Project progress is determined by the scoring of reproducible point values as established by **C 02** Problem-Based Functional Size Measurement Method for the delivery of desired software products.

This speed measure of delivered value can be traced back to an approved definition of which product is to be "done", represented in a prioritized backlog, such as represented by **C 05** Problem-Based Project Baseline. This makes it possible, and is assisted by **C 04** "One4All" View Model on Software Architecture, to replay or revise respective decision making and development plans as needed, and thus to take advantage of lessons learned and to benefit from best practices.

The Problem-Based Speed Benchmark as developed in this dissertation is comparable across projects and teams, since it is normalized on the basis of **C 01** Problem-Based Functional Size Measurement Patterns. These enable to share a common understanding about software product requirements [226, page 351, chapter 18] and respective performance data gathering, since they provide an instrument to reproducibly determine point values for a defined scope of product requirements.

The Problem-Based Speed Benchmark substitutes the Planning Game [55, page 56], an agile project guesstimate practice to judge the work volume of a software project team, by an approach to early functional size measurement. This makes it possible to plan for value delivery at a sustainable pace [214], one within which a project team has demonstrated to be able to meet user expectations. Knowing their speed endorses the team to commit to project plans only, which are ready-to-succeed.

Chapter 9 elaborates this contribution **C 06** Problem-Based Speed Benchmark, which addresses research question **RQ 3.b** How to benchmark the progress of projects?

4.2. Publications

This dissertation compiles parts of its author's research in the field of pattern-based software engineering, which is aimed at putting strategic reuse into software project practice. In this context, the author's research is concerned with problem analysis and its reasonable intersection with solution design to provide for accelerated production of quality software.

This section follows the advice from Parnas (2007), who stated that "If you get a letter of recommendation that counts numbers of publications, rather than commenting substantively on a candidate's contributions, ignore it." [173, page 20], which is why this section discusses the author's scientific contributions in already published research works summarized in table 4.2. It lists all publications relevant to this dissertation, which are co-**authored by Ina Wentzlaff**, and which are refined or newly formed in the context of Problem-Based Project Planning in Postmodern Software Engineering.

Publication	Comment on contributions
[68]	Isabelle Côté, Denis Hatebur, Maritta Heisel, Holger Schmidt, and **Ina Wentzlaff**. A Systematic Account of Problem Frames. In Proceedings of the 12th European Conference on Pattern Languages of Programs (EuroPLoP 2007), pages 749–767, Irsee, Germany, July 4-8, 2007. Universitätsverlag Konstanz. URL `http://www.uni-due.de/imperia/md/content/swe/papers/2007europlop.pdf` This publication comprises the following scientific contributions, which develops the fundamentals of problem frames further: [68].i introducing a new domain type, [68].ii characterizing domain types by their interface behavior, [68].iii exploring problem frames by permutation of domain types. **Ina Wentzlaff** raised the idea to contribution **[68].iii**. Her intent was to understand, why Jackson (2001) in [128] builds his pattern-based approach to requirements engineering on five Basic Frames only. **Ina Wentzlaff** has worked with the other authors of this publication to discuss and write the research work presented. In 2007 she participated in the EuroPLoP Conference in Kloster Irsee, Germany, on behalf of all authors of this publication. This publication is fundamental to all scientific contributions of this dissertation. It is discussed and refined in depth in chapter 5 Problem-Based Units of Measure, and especially considered in section 5.6 Problem Pattern – Frames Revisited for establishing **C 01** Problem-Based Functional Size Measurement Patterns.

Publication	Comment on contributions
[223]	**Ina Wentzlaff**. Establishing a Requirements Baseline by Functional Size Measurement Patterns. In First International Workshop on Requirements Prioritization and Enactment (PrioRE'17), CEUR Joint Proceedings of REFSQ 2017 Workshops co-located with the 23nd International Conference on Requirements Engineering: Foundation for Software Quality (REFSQ 2017), Essen, Germany, February 27, 2017. URL http://ceur-ws.org/Vol-1796/priore-paper-1.pdf This publication presents the following scientific contributions for stabilizing requirements estimates: [223].i introducing a set of basic functional-size measurement patterns, [223].ii presenting a requirements sizing method (Frame Counting Agenda, in table 6.2 on page 82) based on the aforementioned set of patterns, [223].iii deducing validation conditions (in table 6.3 on page 86) from IFPUG IFPUG [117] to safeguard the requirement estimates obtained by the proposed method. **Ina Wentzlaff** is the author of each of these contributions **[223].i**, **[223].ii**, and **[223].iii**, which origin from her independent research activities in this field. She has participated at the PrioRE 2017 Workshop in Essen, Germany, where she presented this publication.

Publication	Comment on contributions
[191]	Holger Schmidt and **Ina Wentzlaff**. Preserving Software Quality Characteristics from Requirements Analysis to Architectural Design. In Volker Gruhn and Flávio Oquendo, editors, Third European Workshop on Software Architecture (EWSA 2006), Revised Selected Papers, volume 4344 of Lecture Notes in Computer Science, pages 189–203, Nantes, France, September 4-5, 2006. Springer. DOI `10.1007/11966104_14` This publication comprises the following scientific contributions allowing for a quality-preserving intertwining of software requirements and design: [191].i illustrating a role-driven mapping of patterns for problem analysis to those of solution design by the example of a chat application, [191].ii considering implementation and interaction of quality characteristics at the level of pattern-based architectural design [191].ii.a from a security perspective (Security Problem Frames [102, 190]), [191].ii.b from a usability perspective (*HCI*Frames [224]). **Ina Wentzlaff** has created Human-Computer-Interaction patterns (*HCI*Frames) out of problem frames based on the idea behind *Arch*Frames [182], thus establishing contribution [191].ii.b of this publication. She developed *HCI*Frames further in collaboration with Markus Specker to make them applicable in usability engineering as presented by Wentzlaff and Specker (2006) in [224] and Specker and Wentzlaff (2007) in [203]. Contribution [191].i has also origin in her research on *HCI*Frames, regarding the use of roles to map patterns for requirements with those of architecture design. **Ina Wentzlaff** has worked with Holger Schmidt to discuss and write this publication. Both authors participated in the European Workshop on Software Architecture 2006 in Nantes, France, and gave a joint presentation on this publication.

Publication	Comment on contributions
[69]	Isabelle Côté, Maritta Heisel, and **Ina Wentzlaff**. Pattern-Based Evolution of Software Architectures. In Flávio Oquendo, editor, Proceedings of the First European Conference on Software Architecture (ECSA 2007), volume 4758 of Lecture Notes in Computer Science, pages 29–43, Aranjuez, Spain, September 24-26, 2007. Springer. DOI 10.1007/978-3-540-75132-8_4 . **Best Paper Award** This publication comprises the following scientific contributions to intertwine requirements and design by means of patterns within an evolutionary software development method: [69].i evolution operators, guiding rework of requirements and design within a pattern-based, evolutionary software development method [67], [69].ii evolution scenarios, illustrating the application of evolution operators by the example of a chat application. **Ina Wentzlaff** has been lead author for contribution [69].ii, and expanded the application of role-driven mapping between patterns for software requirements and design, which has been started by Schmidt and Wentzlaff (2006) in [191]. She has worked jointly with Isabelle Côté and Maritta Heisel to discuss and write this publication, and in 2007 she participated in the European Conference on Software Architecture, Aranjuez, Spain, for presenting this publication on behalf of all its authors.

Publication	Comment on contributions
[70]	Isabelle Côté, Maritta Heisel, and **Ina Wentzlaff**. Pattern-Based Exploration of Design Alternatives for the Evolution of Software Architectures. International Journal of Cooperative Information Systems (IJCIS), 16(3/4):341–365, September/December 2007. DOI 10.1142/S0218843007001688 Impact factor (IJCIS): 0.528 (2009) This publication extends the research of Côté et al. (2007) as presented in [69], and comprises the following new scientific contributions with respect to pattern-based, evolutionary software development: [70].i extending background and detail on evolution operators and their use within a pattern-based, evolutionary software development method, [70].ii extending evolution scenarios by presenting more solution design alternatives and a greater range of involved patterns, [70].iii illustrating the use of software life-cycle expressions to generate, transform and choose among design alternatives. **Ina Wentzlaff** contributed to [70].ii and [70].iii by preparing the ideas and models for the design alternatives and by synchronizing these with the software life-cycle expressions. All authors of this publication have collaborated in discussing and writing the research work presented. Part III Problem-Based Project Adaptation of this dissertation elaborates on the scalability of problem frames in connection with its use and contribution to establish a software architecture design, as has been announced in the future work section of this publication. In addition, this dissertation provides more details on the relevance of software life-cycle expressions, and a role-driven mapping for patterns to seamlessly bridge problem analysis and solution design.

TABLE 4.2 Published scientific contributions

4.3. Limitation

The author's research discussed in scope of this dissertation is directed to functional user requirements for software applications in the domain of information systems.

The conceptions used and introduced in this dissertation require adaption to properly address non-functional requirements or software applications for embedded and real-time systems.

4.4. Structure

As outlined in table 0.1 on page ix, the following paragraphs give an overview of the structure and topics discussed in this dissertation to answer the research questions **RQ 1** to **RQ 3**.

Part I. Software Projects – Perspectives on a Managed Engineering Discipline

Part I introduces the vicious cycle of software projects, which is addressed in this dissertation. One the one hand, there is the management perspective in each software project, responsible for establishing a plan for value delivery. On the other hand, there is the engineering perspective in each software project, responsible for producing a software product, which progresses to plan. Software projects get into trouble, each time the rate of progress (project speed) is not matched with the delivery of value (project size) as planned. Project teams which are in the position to balance these trade-offs are also in control of making their project a success. Chapter 1 Motivation – Empowering software project teams to move faster, presents what speed means to software projects, and why it is challenging to software project processes and teams. It derives research questions to be answered by this dissertation, indicating the gaps to be closed, in order to take advantage of speed control. Chapter 2 Background – On the emergence of product requirements and project teams, discusses the factors that impact project success, and its dependence on plans, which anticipate change. Chapter 3 Research Objective – Sustaining decision making and development by pattern practices, gives insights on the bond of project speed and project size with regard to a defined scope of software product requirements. It proposes the use of pattern practices to sustain both management and engineering tasks in software projects, and which enforce a stabilized project scope that is subject to change control. Chapter 4 Overview – Introducing pre-defined units for planning scope and speed of software projects, summarizes the contributions made available in this dissertation to provide software project teams with a common basis to plan for delivering value and its involved development work both by referring a defined project scope.

Part II. Problem-Based Project Estimating

Part II Problem-Based Project Estimating is about answering **RQ 1** How to $determine$ speed? It develops the conceptions and methods for determining the size of a defined project scope, which is given by a set of software product requirements, in a reproducible way on the basis of patterns. Problem-Based Project Estimating streamlines the input to software project planning. Chapter 5 Problem-Based Units of Measure combines problem-oriented requirements engineering with early software size measurement known as function point analysis, for designing functional size measurement patterns out of problem frames. The resulting Problem-Based Functional Size Measurement Patterns are tailored for facilitating the classification of software product requirements into a measurable problem scope. They serve to establish Requirements Work Packages, which are units of recognizable software product requirements. Chapter 6 Problem-Based Estimating Method elaborates a requirements sizing method following the measurement process given by the International Function Point Users Group standard ISO/IEC 20926:2009. It makes use of the Problem-Based Units of Measure as developed in the previous chapter 5, serving as reusable "proxy" for executing the counting process, which is documented as Frame Counting Agenda. The requirements sizing method is equipped with validation conditions to safeguard its accordance with the standard, and to empower different estimators in determining function points for Requirements Work Packages consistently.

Part III. Problem-Based Project Adaptation

Part III Problem-Based Project Adaptation is about answering **RQ 2** How to $adjust$ speed? It develops the conceptions and methods for determining 'instant options for action' namely design alternatives, which provide the project team with a credible route of development work to be "done" within the project timebox, for satisfying a defined scope of software product requirements. Since Problem-Based Project Adaptation relies entirely on pattern practices, decisions made and development options planned for producing desired project deliverables can be replayed and revised by the team as needed to impact their software project speed. Chapter 7 Problem-Based Units of Work refines the concept of transition schemas known from structured analysis to Transition Templates, for establishing a link between patterns of software problem analysis and those of software solution design. Transition Templates assemble instant models for exploring design alternatives that fit a defined scope of software product requirements, such as given by Requirements Work Packages. Chapter 8 Problem-Based Adaptation Framework develops the 4+1 view model on software architecture further, such that it benefits from Transition Templates, as introduced in the previous chapter 7, and operates on patterns exclusively. The resulting "One4All" view model on software architecture guides the project team in establishing units of work, which provides them with a blueprint or plan for the anticipated delivery of desired working software. On the basis of patterns, the "One4All" view model stabilizes the leeway for the fulfillment of Requirements Work Packages. It eases the integration of recurring development problems to predetermined technology platforms, and supports work plan prioritization according to the software product life-cycle and its projectable value delivery. The enhanced anticipation of development plans and the improved adaptability of requirements fulfillment are two cruicial points of controlling project speed.

Part IV. Problem-Based Project Benchmarking

Part IV Problem-Based Project Benchmarking is about answering **RQ 1** How to $compare$ speed? It integrates Problem-Based Project Estimating and Problem-Based Project Adaptation to an agile project process framework introduced as A S.M.A.R.T. Scrum-A·Gen^{E}DA for taking advantage of a project plan (project backlog) that builds on point values (product size) as measurement for a defined set of software requirements (product scope). This kind of project plan serves as baseline for establishing benchmarks (project speed), which are comparable among projects and teams. Chapter 9 Problem-Based Project Baseline and Speed Benchmark details the use of Requirements Work Packages and the "One4All" View Model on software architecture as developed in this dissertation to set up units for measuring project work progress. Applying these for establishing a work plan of software product requirements and its involved performance baseline for a project, ensures the comparability of speed benchmarks. Benchmarking a Problem-Based Project Baseline – A sustainable planning game demonstrates the use of these units for empowering software project teams to benchmark their project success (points scored) compared to their project plan (points committed), whenever a project timebox is completed. Conducting agile projects by A S.M.A.R.T. Scrum-A·Gen^{E}DA implements problem-based project planning, which comes with built-in means for requirements prioritization and for exploring alternative solution designs. In addition, it supports software project teams in adjusting their decision making and development activities as needed, both proactive and retrospective. Ultimately, it makes sustainable control of software projects possible, for and by a demonstrable delivery of value.

Part V. Case Studies

Part V Case Studies presents two comprehensive application examples for illustrating the use and importance of the contributions given by this dissertation, to control speed of software projects. Chapter 10 Vacation Rentals Web Application revisits a didactic play from the lecture Software Technology, which is used by the Working Group Software Engineering at the University of Duisburg-Essen to demonstrate the ADIT procedure, an agenda-driven and pattern-based software development process. Chapter 11 Student Recruitment Web Portal applies the contributions of this dissertation to a software application, which has emerged from a student project and is used by the Faculty of Engineering at the University of Duisburg-Essen in support of their "International Studies in Engineering" program.

Part VI. Epilogue

Part VI Epilogue compiles the findings and implications of taking advantage from pre-defined units for planning the scope and speed in software projects as investigated by this dissertation. Chapter 12 Conclusion summarizes the answers to the research questions as detailed in section 1.2. Chapter 13 Future Prospect outlines remaining issues and newly found directions for paving the way for worthwhile research, one which contributes to a sustainably managed software engineering discipline in software development projects.

Part VII. Appendices

Last but not least, Part VII Appendices provides supplementary materials to this dissertation. Appendix A ISO/IEC 20926:2009 Complexity and Size Tables belongs to the input documents of the Frame Counting Agenda, specifying the complexity parameters and point values that can be assigned to a Requirements Work Package. Appendix B Sanity Checks intends to justify the quality and fitness of the requirements sizing method proposed by the frame counting agenda to the standard ISO/IEC 20926:2009 and to the certification practices of the International Function Point Users Group. Appendix C Listing of Philosophies represents a loose collection of philosophies around agile project practices. Appendix D Overview on Architecture Design Patterns enumerates commonly known patterns applicable to software architecture design. Appendix E Structures of Architecture Design Patterns lists the structure of those architecture design patterns, which are discussed in chapter 7.3 Transition Templates – Making problems absorb into platform for the development of solution templates. Appendix F For Further Discussion presents a collection of notes on the further development of the contributions in this dissertation. A List of Tables, List of Figures, and List of Examples are complemented by an overview of Acronyms frequently applied in this dissertation. Finally, a bibliography comprising all References used to this dissertation are offered to its dear reader.

Part II.

Problem-Based Project Estimating

*Part II Problem-Based Project Estimating is about answering **RQ 1** How to determine speed? It develops the conceptions and methods for determining the size of a defined project scope, which is given by a set of software product requirements, in a reproducible way on the basis of patterns. Problem-Based Project Estimating streamlines the input to software project planning. Chapter 5 Problem-Based Units of Measure combines problem-oriented requirements engineering with early software size measurement known as function point analysis, for designing functional size measurement patterns out of problem frames. The resulting Problem-Based Functional Size Measurement Patterns are tailored for facilitating the classification of software product requirements into a measurable problem scope. They serve to establish Requirements Work Packages, which are units of recognizable software product requirements. Chapter 6 Problem-Based Estimating Method elaborates a requirements sizing method following the measurement process given by the International Function Point Users Group standard ISO/IEC 20926:2009. It makes use of the Problem-Based Units of Measure as developed in the previous chapter 5, serving as reusable "proxy" for executing the counting process, which is documented as Frame Counting Agenda. The requirements sizing method is equipped with validation conditions to safeguard its accordance with the standard, and to empower different estimators in determining function points for Requirements Work Packages consistently.*

5. Problem-Based Units of Measure

5.1. Introduction

This chapter is about designing *functional size measurement patterns* out of problem frames, which serve within the requirements gathering or analysis of software projects to establish unified requirements work packages. These newly developed patterns are equally suitable for classifying requirements into known problem classes and for measuring their functional complexity, which is given as a point value.

Within the subsequent project planning, the resulting requirements work packages, of which each is then equipped with a now reproducible point value, represent a work item. It can be scheduled to be done in any upcoming project iteration, and paves to way to find candidate solutions for the problem at hand on the basis of patterns.

To create this joint capability of requirements classification and its combined measurement, that allows for utilizing a requirements work package as a pre-defined unit of measure for requirements, section 5.2 Background gives a brief overview of concepts, which are developed further in the following.

Section 5.3 Problem Unit – Requirements Work Package identifies the constituent parts of problem-oriented software engineering and functional size measurement, that are expected to establish a proper match of requirements classification and measuring conceptions. Their specific use to build desired requirements work packages, which form reusable problem units, is then thoroughly investigated in subsequent sections of this chapter.

Section 5.4 Problem Class – Kind of Functionality elaborates the kind of functionality addressed by a requirements work package. It represents an important parameter that impacts a problem's functional complexity, and thus must be addressed by a requirements work package.

The kind of functionality involved with some requirements strongly relates to their associatable problem class. It also determines the rules that apply for estimating the size of a requirements work package. This parameter is approached by introducing three types of functionality, which equally matter to requirements classification on the basis of patterns and requirements sizing.

Section 5.5 Problem Scope – Amount of Functionality elaborates the amount of functionality that is involved with a requirements work package. It represents an another important parameter that impacts a problem's functional complexity.

The amount of functionality involved with some requirements strongly relates to the scope of the requirements' problem class, i.e. how many requirements are in one package. To properly control this parameter, a constant level of detail must be maintained for each problem under consideration, in order to obtain consistent size measurements for each requirements work package. This parameter is approached by customizing a given hierarchy for functional size measurement patterns, and correlating problem patterns with each level of that hierarchy. That way, requirements work packages with a defined level of detail, i.e. with a known functional scope, are achievable.

Since class and scope of a problem, which characterize a defined set of requirements, have been identified as the key parameters for determining their functional complexity, section 5.6 Problem Pattern – Frames Revisited revisits former research into the fundamentals of problem frames [68]. It examines the commensurability of the problem patterns discussed there to set up units of measure for requirements. As a conclusion and final contribution of this chapter, section 5.6 presents a set of *problem-based functional size measurement patterns*, that provision for unified requirements work packages. These newly developed conceptions are placed and discussed in the context of related work in section 5.7 Discussion & Related Work.

Section 5.8 Summary summaries the objectives met in this chapter with regard to the need and contribution of problem patterns to establish measurable units of software requirements.

5.2. Background

This section gives the state-of-the-art literature and concepts used in the following for developing problem-based functional size measurement patterns.

It starts with an overview of some conceptions for grouping software requirements into meaningful units, which must be managed during a project in section 5.2.1. It focuses on Problem-Orientation and Requirements Engineering.

Section 5.2.2 Early Software Measurement introduces those approaches to early software size measurement, which have relevance in ongoing industrial practice, and especially to this work. It considers the role of point values for estimating the amount of functionality involved with software requirements in project planning.

Section 5.2.3 presents the IFPUG FSM Method ISO/IEC 20926:2009 – Terms and Definitions used by the International Function Point Users Group (IFPUG) in its standard ISO/IEC 20926:2009 [117] for functional software size measurement. It is the core instrument used in here.

Finally, section 5.2.4 Categories of Functional Size Measurement Patterns refers a hierarchy of functional size measurement patterns [212], which is applied to problem-oriented software engineering [100, 128] and further refined within a pattern-based approach to early software size measurement as is proposed by this work.

5.2.1. Problem-Orientation and Requirements Engineering

Manifold approaches to requirements engineering such as goal-, object-, feature-, or aspect-oriented ones, etc. exist. They have in common, that they are all about requirements identification and representation, by grouping these in one way or the other into families [41], which exhibit commonalities and variabilities [64] among the requirements. These approaches differ in their requirements (de)composition, e.g. by hierarchy, shared properties, or purpose, for managing involved complexity, and for establishing their refinement to increase requirements understanding.

Compared to later software design or programming, these different approaches share the idea of modularization [171, 172] by bringing it to the domain of software requirements analysis. Ultimately, all of these approaches aim at relations(, sets or structures, that crystalice in black-boxes [26, page 27]) of highly cohesive and low-coupled software requirements, which makes their dependence clear, often supported by a semi-formal, graphical notation, and which is to the ease of their management.

Problem-Orientation makes allowance for the fact, that not every new demand requires technological innovation or original software production. In the broader sense, it enables the requirements analyst to separate the requirements which demand creativity from those that belong to standard development procedures. It integrates the consideration of the real problem to be solved, e.g. of what a user wants, which is stated by requirements, and the potential approaches to its solution, e.g. of what the software does, which is stated in a specification and implemented by software, based on a requirements reference model [98] and accordant refinement framework, therewith establishing a means for systematic reuse and comparability of software development artifacts and project endeavours.

In Problem-Oriented Software Engineering (POSE) [100], requirements are reduced to classes of known problems, which “often have several possible solutions, of which some could be very hard or expensive to achieve” [140, page 215]. This process is supported by patterns for representing these problem classes named Problem Frames [128], which are giving a form to a requirements statement and thus become utilizable as requirement templates. Different styles and techniques to build templated requirements statements by following an approach to POSE [138] are available, which have proven the use and relevance of problem-orientation in practice [139]. In case a recurring problem has been identified, a former (way and the decisions made to produce or dismiss its) solution can be taken into account. It is expectable to gain effort and quality benefits by reuse of respective artifacts, which already have demonstrated their effectiveness.

User Stories

In agile software development and in a project planning that makes use of Planning Poker [54], requirements are usually enclosed to user stories. These have origin in extreme programming and represent a brief, informal description of functional software-user interaction. In this regard, user stories are comparable to UML use cases [168]. By contrast with use case diagrams, which are not self-explanatory and thus must be evolved by scenarios, user stories have to be complemented [21, page 228] by additional information, too, such as acceptance criteria (definition of "done") and value indicators (story points), for only naming a few. Each user story is seen as a token or unit of work for stimulating conversation in the team to reach consensus on what is wanted by the user. Accordingly, Jeffries [129] defines the three components of a user story as the three C's: Card, Conversation, and Confirmation. User stories are documented in a canonical format, i.e. by following a template for representing a requirements statement. They are usually written on index cards or post-it notes for use at the kanban/project status board, or respectively, for quickly handing them on among the members of a project team for working on these. In agile, “what happens around the user story is far more important than the user story itself” [21, page 231].

Templated Requirement Statements In order to care for well-written user stories, these often follow a simple format, such as given by Cohn [60] and in [21, section 25.6.1.2]. For example, the template:

“As a <role>, I want <function> so that <benefit/some reason>”

describes the who, what, and why of a software requirement, and helps in creating uniform requirement expressions. It leaves open, how much detail is appropriate or must be added to obtain meaningful statements. It does not define the limit for “just enough” requirements. The obtained statements should be likewise small and separated enough to provide for flexibility in their planning and in executing independent work on these.

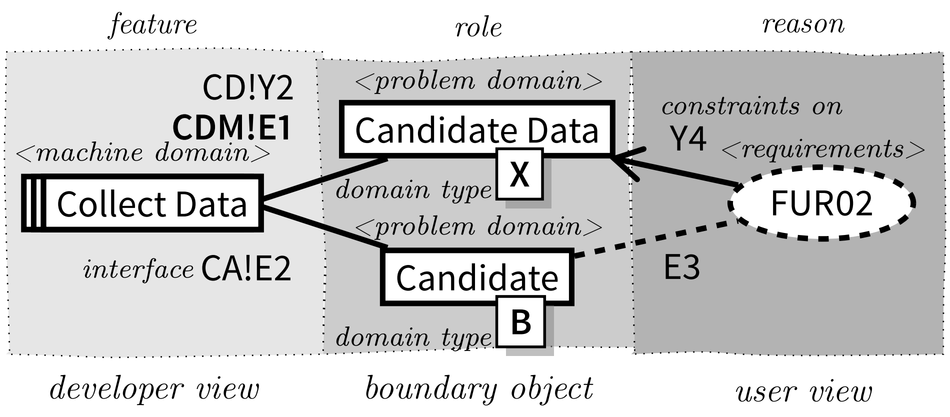
Further criteria to decide on good requirements are the INVEST mnemonic of requirement statements developed by Wake [216], which Waters [221, pages 137–147] also references, and in addition, the SMART citeria as introduced by Doran [82] and discussed by table 3.2 on page 21 are applicable.

What remains, is the level of granularity to be clarified for making a user story a team can work on [197]. Since initial requirements are often vaguely defined, these are handled as a high-level user story named epic (or just a level further as a so-called theme [57]), which will be broken down over time [21, page 229]. It is an open issue, what means enable to separate effectively an epic from a user story in regard to their level of detail. Supporting this decision making in a systematic manner, which is tried in here by utilizing patterns, would be very welcome to the engineering and management of software project work.

Problem Frames

Problem Frames [128] are a tool used in software engineering, which assists the requirements analyst in classifying functional software requirements into classes of simple, recurring problem situations. Therefore, each problem frame is given by a templated structure called frame diagram, that is filled in with the software requirements by the analyst. This results a model of software requirements, which not only has been formed upon patterns for known problems, but which also supports the analyst in deriving respective software specifications [128, page 106] in a reproducible way. Its the objective of the problem frames approach to cope with problem complexity in requirements engineering. This approach has been developed by Jackson [126] since 1995. By its use, the overall problem to be solved that is given by a set of software requirements, is projected into simple problems, which can be considered independent of each other. Using frame diagrams for requirements analysis, establishes a problem or respective requirements decomposition by separation of concerns. The advanced meaning and value of these (frame) concerns, which are inherent to each problem frame, is part of the contributions made by this dissertation. It is therefore investigated in detail in the following sections, whereby section 5.4.1 on page 50 and the next paragraph about frame diagrams, introduce their general use.

Frame Diagrams For each problem frame a diagrammatic notation named frame diagram can be created. In this regard, the terms problem frame and frame diagram are often used interchangeably. Each represents a unique combination of problem domains and their respective shared phenomena, where both can differ in type, quantity and correlation. Figure 5.1 outlines a frame diagram introduced by Jackson [128, page 96, section 4.3.4], which is named "simple workpieces" problem frame. Its meaning and annotations is described here in brief.



Legend of interfaces E1 to E3, Y2, and Y4 and their {shared phenomena}:
E1{store40FormData}, E2{fillIn40FormData,FormData1..40}, E3{collectCandidateData}, Y2{FormData1..40}, Y4{collectCandidateData}

FIGURE 5.1 Annotated "simple workpieces" problem frame, cf. [223, fig. 2]

On the left hand side of figure 5.1 is the machine domain, which represents a part of the software application to be built. It is indicated by a box with two vertical bars. In the middle of this figure are two problem domains, indicating entities involved with the problem to be solved. Each problem domain has a particular domain type, here biddable (B) and lexical (X). Section 5.4.1 on page 50 and table 5.4 on page 70 give more details on which domain types exist, and their definitions. The machine and the problem domains are connected via interfaces, e.g. E1 to E3, Y2, and Y4. Interfaces are represented by solid lines between domains and hold so-called shared phenomena. These represent the information and interaction shared by domains that are interfaced with each other. An exclamation mark indicates the domain, which has control of what phenomena. On the right hand side of figure 5.1 are the requirements, which are of relevance to the problem discussed by this frame diagram. The set of respective requirements is addressed by an dashed oval.

The general concern depicted by a frame diagram is that the software to be built (on the left) has to control a specific part of the problem (in the middle) in a way as the requirement demands [128, page 107] (on the right). A problem frame indicates this part by a requirement constraint on the respective problem domain, which is an arrowhead that points to the respective problem domain. Further problem domains that contribute (by means of shared phenomena) to this problem, i.e. that belong to the problem context, are connected with the requirements (oval) via dashed lines, that mark a requirements reference.

The frame diagram in figure 5.1 is an instance of the simple workpieces problem frame for a requirements statement named FUR02. In this example, FUR02 demands to support an application procedure by a recruitment software (machine) that is to be built. The desired software has to collect data from a candidate (E3) and to make it accessable for later use (Y4) here by storing these (E1). This candidate data (Y2) is provided by a candidate via filling out respective forms (E2), which must be developed for and provided by the recruitment software. It can be said, that the general problem modeled by usage of a simple workpieces problem frame is always about or concerned with recording and including the alteration of some provided information.

Its the configuration of problem domains in their number, type and (constrained) reference to requirements which characterize a problem frame and make their unique frame concern. If a requirement maps a particular problem frame, it is reasonably supposable that this requirement belongs to a known problem class. That way grouped requirements are expected to be solvable by comparable means, which is a chance to reduce development effort and to care for quality by reuse of artefacts that already have demonstrated their value. The contribution of this chapter 5 Problem-Based Units of Measure is to elaborate and discuss how to properly limit the number of configurations (or possible frame permutations) to a helpful extent.

In practice, problem frames can be comparably applied to user stories as templates or templated structures for gathering requirement statements. Figure 5.1 shows how the Connextra format maps the constituent parts of a frame diagram by a *role*, *feature*, and *reason* part. It is a commonly used template for user stories and " highlights the <u>who</u>, the <u>what</u> and the <u>why</u>" [149] of a functional user requirement. It is an interest of this work to reach synergetic effects that enhance requirements communication between developers and users. The intention is to leverage a teams' shared requirements understanding by merging these two approaches of requirements template use, i.e. stories and frames, into an unique symbiosis.

However, the acceptability of this procedure is subject to ongoing discussions [9, 149]. Concept confusion in regard to the meaning of requirements along their spectrum of business goals to technical specifications [140] must be resolved, and also the different use of requirements documentation for planning a software development project in either the traditional (plan-driven) or in the agile (process-driven) sense is the other intrinsic challenge to overcome in this context.

Figure 5.1 on page 40 gives an idea why problem frames are a proper means for serving as a kind of pattern-enhanced "boundary objects" [26, 145] that bridges the different perspectives on requirements in a project team. These templated structures allow to join the views of developers and users on the software requirements into a shared perspective, instead of creating a cascade of isolated abstractions, that yield unfavorable handovers, and thereby widens involved semantic gaps on the requirements.

5.2.2. Early Software Measurement

Early software measurement is about estimating software product requirements in the beginning of a project. It is in contrast to software measurement in later project phases, where some code or a prototype for a software product is available, which then undergoes the measurement procedure.

Different measurement methods and metrics exist. For instance, lines of code (LOC) is one popular metric, which has application in many measurement approaches, although its expressiveness heavily relates to the employed development environment, regarding the technological platform and individual developer. This makes a comparison or the conversion a.k.a. backfiring of LOC to other metrics difficult, often “rarely useful” [114, page 89], and is even seen as “professional malpractice” [131].

Technology/ist-agnostic, model-based approaches help to overcome these troubles, by quantifying or counting the conceptualized characteristics of a software, which makes them applicable in early as well as in late project phases, and thus attractive for benchmarking [192] and accordant decision-making [166, page 7, section 6.2] in project planning. For instance, function points is a metric used within function point analysis for determining the functional size of a problem, which is representable by a requirements specification. Function points serve in equal measures for expressing the functional size of a solution, which may exist as a prototype, or is given as concept or by code of a software product.

Figure 5.2 shows, that transforming requirements to point values requires the application of a counting regime to the determinants of size, one which provides the models and rules to assess the relevant software characteristics, for obtaining size measures, or requirement estimates, respectively. If these software characteristics are determined reproducibly and become comparable, so are the results of the measurement method, when following the same counting regime.

This is the challenge addressed by subsequent chapters.

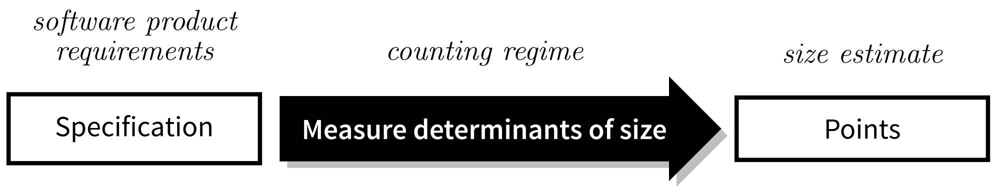


FIGURE 5.2 Transforming requirements into points, adapted from [175, page 31, figure 3.2]

Function Point Analysis is a functional size measurement method [92] for software, which provides a product metric in the early phases of software projects for determining the size of a desired piece of software. It is a technology-agnostic approach, which classifies functional, user-recognizable requirements into logical so-called base functional components to be counted within the sizing process. Function Point Analysis is no Requirements Engineering method, since its prime purpose is not to result a comprehensive requirements specification. Actually, the quality of Function Point Analysis depends on a good requirements specification [154]. This is the reason for combining function point analysis with problem-oriented software engineering [100] in the following. Function Point Analysis introduces a means for quantifying software product requirements specifications, which significantly contributes to their comparability, and which is a valuable input to effort estimation in the software project planning process.

Function Point Analysis was originally created by Albrecht (1979) as presented in [2], who suggested a measure for application development project productivity. It allows for calculating the functional size of an application as a number given as so-called unadjusted Function Points. This point value represents a size measure, which is solely based on the amount and location of data and the interaction needed to process it and thus independent of any technology considerations.

System or quality characteristics, which in early versions of Function Point Analysis have been addressed by a value adjustment factor, are today managable via non-functional size measurement given for instance by the Software Non-functional Assessment Process, SNAP [119]. It provides a complexity adjustment of an unadjusted function point count, which yields in adjusted function points. SNAP is not considered further, but it represents the decisive reason for using the IFPUG counting regime introduced in section 5.2.3 and for applying this specific measurement standard throughout this work.

Planning Poker as coined by Grenning (2002) in [97] and popularized by Cohn (2005) in [55, chapter 6, page 56] is a consensus-based estimation technique for producing relative size estimates to brief requirement statements. It has become established in agile project practice, where it is also known as Scrum Poker or the Planning Game. This collaborative forecast method is a modern adaption of the Wideband-Delphi estimation technique [34, Page 335ff] used for software size estimation since the 1980s.

The requirements are enclosed in user stories [54], each is to be assigned with a point value by the members of the project team. By playing Planning Poker, the team justifies the amount of functional scope expected to be required for the delivery of a user story, giving respective point values to it. To this, Planning Poker makes use of a predefined scale based on a fibonacci sequence. It provides the team with a standard of assignable numbers, named story points, which serve to express the size of a user story [55, page 36]. This scale is usually given by a deck of cards to each team member. The members of the team start estimating each user story individually by arranging these along the scale. Then the cards are revealed, and the group seeks to reach consensus on these individual relative size ratings by discussion.

The choice of these "magic (size) numbers" as both magnitude for the scale as well as size for a particular user story is made arbitrarily. The precision and consistency of estimates resulting from Planning Poker and its therewith involved reproducability of size estimates relies on a well-rehearsed team [220], whose consensus reached is supposed to be of higher reliability [146] than that of an unstable one. This has proven to be a viable approach to small projects, but it is hardly a possible one for large-scale projects requiring careful coordination.

5.2.3. IFPUG FSM Method ISO/IEC 20926:2009 – Terms and Definitions

Figure 5.3 gives an overview of the "determinants of size" inherent to a software (requirements specification, compare figure 5.2 on page 42), which form the basis for executing Functional Size Measurement (FSM) by the IFPUG standard ISO/IEC 20926:2009 (ISO 20926) [117], and which enable the assignment of point values to the countable software characteristics.

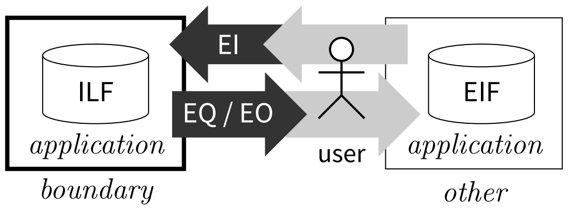


FIGURE 5.3 Overview of base functional components measurable by IFPUG [117]

Each FSM depends on the proper establishment of the **application boundary**, which decides on the software characteristics that need to be counted. It is a "conceptual interface between the software under study and its users" [117, page 3, section 3.9]. The **user** is in this connection a "person or thing that communicates or interacts with the software at any time NOTE 'Things' include, but are not limited to, [other] software applications, [...], sensors and other hardware" [117, page 7, section 3.50]. From the perspective of problem-oriented software engineering, the application boundary conforms to the software requirements specification, describing the interactions between the software-to-be and its environment. Against this background, it is obvious that **functional user requirements** (FUR) are in the focus of functional size measurement. They represent a "sub-set of the user requirements specifying what the software shall do in terms of tasks and services" [117, page 5, section 3.34]. Interactions and involved data that cross the application boundary represent the determinants of size in Function Point Analysis. As illustrated in figure 5.3, each out of the five base functional components (ILF, EIF, EI, EQ, EO, which are presented in the next lines of text) is counted with respect to the application boundary, which makes their determination crucial for the outcome of a FSM. In order to determine the functional size of some requirements consistently, these need to be decomposed to fit the base functional components in a reproducible way. A **base functional component** (BFC) is defined as an "elementary unit of FUR defined by and used by an FSM Method for measurement purposes" [117, page 2, section 3.8]. So what a BFC looks like, depends on the requirements specification approach. By the IFPUG standard IFPUG [117], a BFC is either a data function or a transactional function.

- **Data Functions** represent "functionality provided to the user to meet internal or external data storage requirements NOTE A data function is either an Internal Logical File or an External Interface File" [117, page 3, section 3.16].
  - An **Internal Logical File** (ILF) is a "user recognizable group of logically related data or control information maintained within the boundary of the application being measured" [117, page 6, section 3.39].
  - An **External Interface File** (EIF) is a "user recognizable group of logically related data or control information, which is referenced by the application being measured, but which is maintained within the boundary of another application" [117, page 5, section 3.29].
- **Transactional Functions** describe an "elementary process that provides functionality to the user to process data" [117, page 7, section 3.49]. Such an **elementary process** describes the "smallest unit of activity that is meaningful to the user" [117, page 4, section 3.21], and it "must be self-contained and leave the business of the application being counted in a consistent state." [118].

There are three kinds of functional or so-called elementary processes, namely **External Input** (EI), **External Inquiry** (EQ), and **External Output** (EO), which are discussed in detail in section 5.4.2 Primary Intent of Elementary Processes on page 51, and again specifies what the software shall do in terms of different kinds of data processing or software functionality.

To this extent, the application of elementary processes for decomposing FUR is comparable to the use of problem frames in the requirements engineering. The next chapters detail the mapping of respective conceptions.

5.2.4. Categories of Functional Size Measurement Patterns

Trudel et al. (2016) introduce in [212] “the concepts of functional size measurement (FSM) patterns”, defining different portions of functional process(es) and involved data groups, as needed for executing functional size measurement by the COSMIC [66] counting regime. According to this publication, “A FSM pattern is a predefined generic software model solving a recurring measurement problem in a specific context”, thereby following pattern usage and description as defined by Alexander et al. [5], and brought to software design by Gamma et al. [90].

Figure 5.4 gives the proposed hierarchy of FSM patterns, for classifying a number of functional processe(s) and data groups involved into different levels of granularity [65, page 30, section 2.4]. These four categories of functional size measurement patterns can be seen as partly embedded.

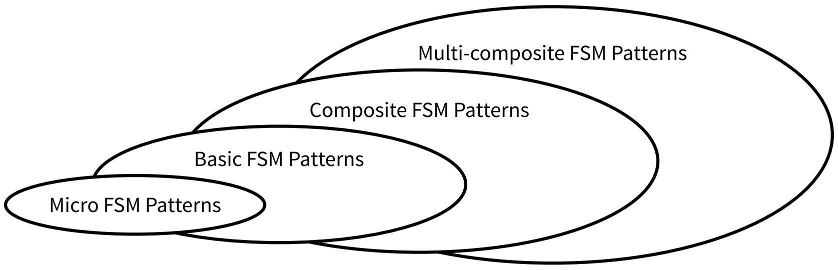


FIGURE 5.4 Hierarchical representation of FSM patterns, taken from [212, figure 3]

In COSMIC [65, page 32, definition], a functional process is a “single event [the software] must respond to”, which is triggered by an “individual [...] functional user”. On this basis, a functional process in COSMIC is comparable to an elementary process of IFPUG functional size measurement, as discussed in section 5.4.2 in detail.

The following paragraphs recapitulate the definition of each FSM pattern given by Trudel et al. [212]. These definitions of micro, basic, composite, and multi-composite FSM patterns are used in table 5.3 on page 55 for leveling problem frames into this hierarchy. This results a means to specify recurring problems in a specific context, accordingly. Section 5.5.1 exemplifies, why Basic FSM patterns become the established standard level of granularity [65, page 32, paragraph 2.4.3] for classifying and measuring requirements. Section 5.6.3 derives problem-based functional size measurement patterns based on the definitions of FSM patterns given here, that belong to exactly one, namely the level of Basic FSM patterns, for satisfying the need for a standard level of granularity [65, page 30, paragraph 2.4.1], in order to identify and measure a defined scope of software product requirements.

Micro FSM Patterns “applies to a fragment of a functional process, involving one or several data groups.”

Basic FSM Patterns “applies a complete yet single functional process [...which] can also handle multiple data groups. [...] a basic FSM pattern is related to a whole functional process.”

Composite FSM Patterns “applies to a set of basic FSM patterns having a high level functional meaning together. Instead of being restricted to a single functional process (the particularity of a basic FSM pattern), a composite FSM combines several functional processes. These functional processes have the characteristic of sharing the same primary data group [...]”

Multi-Composite FSM Patterns “applies to a set of composite and basic patterns having functional relationships among them. A multi-composite FSM pattern combines multiple functional processes handling several data groups within the software being measured.”

5.3. Problem Unit – Requirements Work Package

A challenge within project planning is to set up commensurable units of requirements, that are meaningful to most members of a project team. Information about desired software functionality, i.e. the requirements, need to be reasonably grouped in a work package, such that it provides the team with a common point of reference in decision making as well as in executing the project process activities [7, page 47]. Establishing units of comparable software development problems, that are recognizable to the team, is expected to provide for consistent requirements estimates and accordant work plans of increased predictive value.

Problem-oriented software engineering and functional size measurement provide reusable means to categorize requirements for classifying their involved functionality and for their measurement. These means, i.e. problem frames as introduced in brief on page 40 and elementary processes as briefly presented on page 43, are synergized to problem-based functional size measurement patterns in the following. They are designed to establish requirements work packages (RWP) as introduced next, that exhibit the joint capability of recognizing and estimating software requirements in a reproducible way. These RWP become meaningful to any member of a project team in the sense of a pattern-enchanced boundary object (PEBO) [26, 145], which supports their collaboration.

DEFINITION 5.1 Requirements Work Package

constitutes a meaningful unit of desired software functionality, which is

- self-contained and
- measurable.

5.3.1. Self-Contained Functionality

As illustrated in figure 5.5, problem frames serve to classify a set of requirements into a complete, i.e. self-contained (sub)problem, representing "A task to be accomplished by software development." [127, page 368]. When analyzing a "subproblem, assume that the other subproblems are solved. [which is ...] the essential basis of any effective separation of concerns" [127, page 60]. Consequently, each subproblem is "having its own requirements and its own problem context." [127, page 371] which forms a defined level of requirements granularity. Thus, a subproblem is a unit of independent and small(-enough to proceed with these in project planning and development) software requirements, documenting what a user wants in terms of changes to the given problem context (conditions) [127, page 369], which are to be achieved by some functionality of the software to be built.

A requirements work package can be considered as unit for classifying requirements and thus as a subproblem, which is created by means of problem frames. It provides a consistent representation for recognizable requirements, which are given by domains and their shared phenomena.

5.3.2. Measurable Functionality

As illustrated in figure 5.5, elementary processes serve to determine the base functional components, which in accordance with ISO 20926 [117] represent the measurable interactions and information of a defined set of requirements. Therefore, so-called transactional and related data functions involved with each elementary process, which are briefly introduced on page 43ff, must be identified for determining the requirements' functional size using the ISO/IEC 20926:2009 Complexity and Size Tables as given in appendix A.

From this perspective, a requirements work package can be considered as a unit for measuring requirements. Within a functional size measurement method, it makes a grouping of requirements in relation to an elementary process possible. It exhibits all measurable base functional components that are of relevance for estimating the functional size of some desired software functionality.

Thus, a requirements work package embodies a known software development problem, whose functional complexity, that is expressible in function points, can be determined in a consistent and reproducible way. This makes a requirements work package a key instrument to benchmark product size and project performance. Part IV of this dissertation looks into this subject.

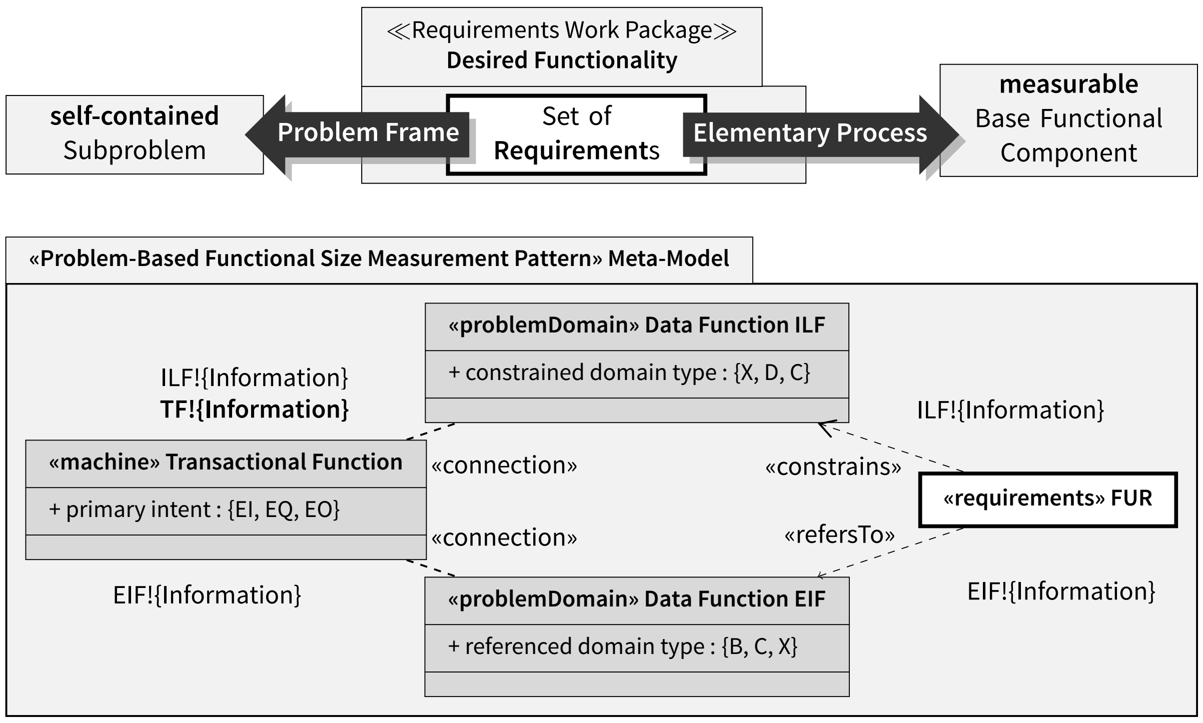

≪Requirements Work Package≫
Desired Functionality
self-contained Subproblem
Problem Frame
Set of Requirements
Elementary Process
measurable Base Functional Component
«Problem-Based Functional Size Measurement Pattern» Meta-Model
«problemDomain» Data Function ILF
+ constrained domain type : {X, D, C}
ILF!{Information}
TF!{Information}
ILF!{Information}
«machine» Transactional Function
+ primary intent : {EI, EQ, EO}
«connection»
«constrains»
«requirements» FUR
«connection»
«refersTo»
EIF!{Information}
«problemDomain» Data Function EIF
+ referenced domain type : {B, C, X}
EIF!{Information}

FIGURE 5.5 Conceptualization of a Requirements Work Package

The quality of this instrument relies on a proper match between problem frames and elementary processes, which is summarized in table 5.1 and illustrated by the meta-model in figure 5.5.

In its upper half, figure 5.5 shows that this match of conceptions is achievable by a requirements work package, which groups desired software functionality to a self-contained subproblem, that on the one hand fits into a recognizable problem class, and on the other hand represents in equal measures the base functional components for this set of requirements, which are of importance to functional size measurement.

In its lower half, figure 5.5 depicts a join of the constituent parts of problem frames and elementary processes that underlie a requirements work package, or more specifically its realized problem-based functional size measurement pattern. The representation of the meta-model makes use of a stereotype notation as applied in the UML4PF eclipse plugin [107] for modeling problem frames.

Table 5.1 contains a legend for the meta-model in figure 5.5 and can be understood as a preview on subsequent sections that explain in detail the constraints on the use of a requirements work package as a problem-based unit of measure for requirements.

Requirements Work Package		Problem Size (Chapter 6)
Problem Frame as unit for classifying requirements · frame concern	**Elementary Process** as unit for measuring requirements · primary intent	Problem Class (Section 5.4)
Domain Types · Problem Domain · Machine Domain	Base Functional Components · Data Function (ILF, EIF) · Transactional Function (EI, EQ, EO)	
Machine Interface involves Shared phenomena · causal shared phenomena · symbolic shared phenomena	Application Boundary involves Information: Data Element Types (DET) · control information · data information	Problem Scope (Section 5.5)
Basic FSM Pattern[1]	Single Elementary Process[2]	

[1] cf. Table 5.3 level II. Basic and Table 5.5
[2] cf. Table 5.2

TABLE 5.1 Constraints on a problem-based unit of measure for requirements

For developing a joint understanding of what a problem class is about, section 5.4 Problem Class – Kind of Functionality establishes a correspondence between problem frames and elementary processes by mapping the conception of frame concern to that of primary intent. It justifies the appropriateness of equating problem domains with data functions, and the machine domain with transactional functions. That way, the same kind of desired software functionality becomes consistently measurable as demanded by section 5.3.2.

Section 5.5 Problem Scope – Amount of Functionality establishes a correspondence between the concepts of machine interface from the problem frames, and the application boundary as defined in functional size measurement to elaborate a mutually meaningful definition of problem scope, which is of use to problem recognition and problem measurement. First, it elaborates the meaning of interaction and information by creating a relationship between the shared phenomena at the machine interface and the data element types, which can undergo function point counting. Second, it clarifies what a basic level for functional size measurement pattern is, and how to make use of it for producing a set of requirements that possess a defined level of granularity, one which comprises and is tailored to a single elementary process only. That way, desired software functionality becomes not only packaged into a self-contained unit of work, namely a RWP as demanded by section 5.3.1, but also its amount and dependence compared to other requirements is made transparent, which increases its objective consideration.

Chapter 6 Problem-Based Estimating Method provides a method, which is based on the conceptualization of a requirements work package as outlined in figure 5.5. It takes the problem-based functional size measurement patterns as listed by table 5.5 on page 72 which result from the meta-model for determining problem size in a systematic and deterministic way.

5.4. Problem Class – Kind of Functionality

A requirements work package is a self-contained and measurable set of requirements meaningful to the project team. Comparably to components of software design, which encompass deliverable software functionality, a requirements work package is a component of software analysis, which comprises a group of desired software functionality. In each case, these components whether representing desired or deliverable functionality are categorizable into recurring “classes of functionality and forms of interaction they provide” [198, page 149].

Consequently and even though with little surprise due to the involvement of problem frames, a requirements work package implements a set of requirements that is concerned with a particular class or kind of functionality.

Knowledge about the kind of functionality is relevant to classify and measure requirements properly. It is likewise a recognition feature of problems and of base functional components, which determines comparability of requirements and the rules that apply for estimating them. It is a fundamental property of a requirements work package, which impacts establishing consistent and reproducible point values for requirements.

DEFINITION 5.2 Kind of Functionality

is a recognition feature of a requirements work package, that determines

- comparability of problems and
- the rules to apply for estimating these.

In order to care for this important characteristic of a requirements work package, which likewise represents an important parameter to a problem’s functional complexity, section 5.4.1 investigates how problem frames and section 5.4.2 investigates how elementary processes are concerned with this issue.

Section 5.4.3 makes use of the findings to combine the constituent parts of problem-oriented requirements engineering and functional size measurement with regard to their involved kind of functionality. Therefore, three **t**ypes **of f**unctionality (TOFF) are introduced to permit a proper categorization and respective combination of conceptions.

5.4.1. Frame Concern of Problem Frames

Problem frames are concerned with different kinds of functionality. In this context, the frame concern as “central concern for problems of a class defined by a problem frame” [128, page 365] is of special interest. When using problem frames to classify requirements, the frame concern helps to identify the kind of functionality that is of relevance to the problem. It determines whether a requirement fits a problem class or not. In case of a match, the machine to be built, i.e. the software application, has to control a specific part of the environment in a way as the requirement demands [128, Page 107].

A problem frame indicates this part by a requirement constraint on the respective problem domain. The frame concern is related to this constrained problem domain, which can be of a le(x)ical, (d)isplay, or (c)ausal domain type, cf. the meta-model in figure 5.5 on page 47. The functionality of a software as describable by problem frames establishes one of the following three types of machine control: a constrained

$\boxed{\text{X}}$ ←-- lexical domain “is a physical representation of data” [128].
The machine controls the read or write operations to these data.

$\boxed{\text{D}}$ ←-- display domain is “an output device for the machine” [68].
On behalf of the machine, it provides “information to other problem domains” [68].

$\boxed{\text{C}}$ ←-- causal domain is provided with information controlled by the machine to invoke specific behavior of this problem domain.

These three types of functionality or respective types of machine control are inherent to each problem frame. They characterize different kinds of processing data or signals in connection with a constrained problem domain, i.e. software functionality, which is needed to fulfill the requirements. These findings are resumed in section 5.4.3 to combine problem frames with elementary processes.

5.4.2. Primary Intent of Elementary Processes

In functional size measurement according to ISO 20926, elementary processes are concerned with different kinds of functionality. In this context, their primary intent is of special interest, which is the “intent that is first in importance” [117, page 6, section 3.43] to each elementary process. The primary intent is related to the kind of processing data or signals, which is visible at the application boundary.

Each elementary process can be distinguished by its primary intent, that characterizes the processing objective or the lead purpose of established functionality with regard to some requirements. There are three types of transactional functions, also known as elementary processes, that allow to classify requirements and serve as base functional components for functional size measurement:

EI ← External Input (EI) is an “elementary process that processes [...] information sent from outside the boundary” [117, page 4, section 3.27], its primary intent is to “maintain an Internal Logical File (ILF) [...]” [118].

EQ ↭ External Inquiry (EQ) is an “elementary process that sends [...] information outside the boundary” [117, page 5, section 3.28], its primary intent is to “present information to a user. It presents only data that is retrieved [...]” [118].

EO → External Output (EO) is an “elementary process that sends [...] information outside the boundary and includes additional processing logic beyond that of an external inquiry” [117, page 5, section 3.30], its primary intent is to “present information to a user. It presents data that is calculated or derived [...]” [118].

The functionality of a software application as describable by elementary processes serves one of these three primary intents. They characterize different kinds of processing data or signals in connection with a set of requirements, that relate to some base functional components. Each provisions for corresponding rules and size values as in ISO/IEC 20926:2009 Complexity and Size Tables in appendix A to measure the functional size of the respective requirements set.

These findings are resumed in the following section 5.4.3 to combine problem frames with elementary processes.

5.4.3. Mapping Patterns to Processes by Types of Functionality

This section combines problem frames and elementary processes with regard to their involved kind of functionality as defined on page 49. Table 5.2 gives an overview of the relationship between these concepts. It shows that problem frames and elementary processes are comparable classes or components for requirements. Both characterize units of software functionality, that exhibit unique kinds of processing some information. These kinds of information processing are distinguishable into categories. In the following, three unique types of functionality (TOFF) are introduced, that allow for comparing problem frames with elementary processes and provide the basis for their proper match.

	unit for requirements	kind of functionality or processing information relates to			
Problem Frame (PF)	problem class	frame concern	$\boxed{X}\dashleftarrow$	$\boxed{D}\dashleftarrow$	$\boxed{C}\dashleftarrow$
Elementary Process (EP)	size class	primary intent	EI $\leftarrow$	EQ $\leftrightsquigarrow$	EO $\rightarrow$
PF and EP coincide with regard to their **type of functionality** (TOFF):			**TOFF-i.**	**TOFF-ii.**	**TOFF-iii.**

TABLE 5.2 Mapping problem frames to elementary processes by types of functionality

DEFINITION 5.3 Type of Functionality

provides a classification of requirements with respect to their involved kind of functionality. It characterizes software functionality that

TOFF-i. processes information, which is received.

TOFF-ii. provides information, which is retrieved.

TOFF-iii. provides information, which is derived.

Types of functionality provide a classification, which ensures that a known problem is measured consistently. As designed here, they provide a bridge between problem and size units for requirements, namely problem frames and elementary processes, with regard to their involved kind of information processing. With respect to the meta-model in figure 5.5 on page 47, TOFF-i. to TOFF-iii. allow for classifying a set of requirements by problem frames that coincides with determining the applicable elementary process. This in turn drives the choice of rules that apply, i.e. the base functional components together with their respective complexity and size matrices, for measuring the problem's functional size. This finding is resumed and central to section 5.6.3, when types of functionality are applied to set up problem-based functional size measurement patterns. In the following, each type of functionality is explained in detail, for providing background to their appearance within the three, right-hand columns in table 5.2 above.

TOFF-i. processing received information

Requirements which belong to TOFF-i. fit to a problem class with a constrained lexical domain. They demand software functionality, that processes some information, which is received and thus not created by the software itself. This type of functionality coincides with an external input in functional size measurement, where information received from outside the application boundary is processed by means of an internal logical file.

For example, reads or writes of a lexical domain $Party\ plan$ are required, in order to process the $Editing\ commands$ received from the users $John\ and\ Lucy$ in Jackson's Party Plan problem, see section 6.5.1.

TOFF-ii. processing retrieved information

Requirements which belong to TOFF-ii. fit to a problem class with a constrained display domain[1]. They demand software functionality, that provides information, which is retrieved. This type of functionality coincides with an external inquiry in functional size measurement, where information is simply retrieved from maybe multiple sources before it is sent outside the application boundary for information purposes.

In contrast to TOFF-iii, requirements which belong to TOFF-ii. are concerned with providing some information to a user or another software application, which are not subject to significant calculations. There is no mentionable data transformation. Retrieved information is taken 'as-is' and more or less forwarded without additional computational effort.

For example, the printout of $sensor\ information$ in Jackson's Local Traffic Monitoring problem is such an 'as-is' representation of data, which is retrieved from the $Vehicles$ that pass sensors on the street, see section 6.5.2.

TOFF-iii. processing derived information

Requirements which belong to TOFF-iii. fit to a problem class with a constrained causal domain[2]. They demand software functionality, that provides some information, which is derived. This type of functionality coincides with an external output in functional size measurement, where information is derived to some distinguishable computational extent, and then sent outside the application boundary for providing information and generating respective control.

In contrast to TOFF-ii, requirements which belong to TOFF-iii. provides information to an other software application, which invokes some control effort to it.

For example, in Jackson's Occasional Sluice Gate problem, $Gate$ status and $Sluice\ operator$ commands must be considered to derive proper machine control of the sluice gate motor, e.g. $On, Off,$ $Clockwise, Anticlockwise$, which requires a status-dependent decisioning of the $Sluice\ controller$ machine, that is to be implemented by respective software, see section 6.5.3.

[1]Note: The fact that a display domain, which has been introduced by Côté et al. [68], indicates one out of three possible problem classes for requirements counting and composition, reassures the relevance of this newly identified domain type in the problem frames theory for analyzing requirements and in requirements measurement practice. This has been hardly demonstrated in literature or research until now.

[2]Note: As already mentioned in section Limitation on page 31 this work focuses on information systems. Due to this reason, a causal domain is seen as another, independent software application, from which the currently considered software (machine) to be built receives some data (symbolic phenomena) or control (causal phenomena) information and vice versa.

5.5. Problem Scope – Amount of Functionality

A functional size measurement pattern as defined by Trudel et al. [212] is “a predefined generic software model solving a recurring measurement problem in a specific context”, which is also presented in section 5.2.4 Categories of Functional Size Measurement Patterns. Problem Frames meet this definition to the extent of being similarly capable to frame a unit of requirements to a recurring problem class in a specific problem context.

As elaborated in the previous sections, a known measurement problem is characterized by the kind of functionality, that it poses. It can be categorized by types of functionality (TOFF-i. to TOFF-iii.), which represent typical ways of processing some information, and which are measurable in a defined way. That is, a recurring problem, which is of a known class, is always measured according to the same rules.

This section addresses the specific context of a problem, which is characterized[3] by the amount of functionality, e.g. the number of distinct data and control information to be managed by the software (machine) to be. This quantity of information essential to understand the problem at hand should be measured in the following. Its size depends on the scope of a problem, and varies with the level of detail applied to the requirements for their analysis or measurement. Stabilizing the scope of requirements or their problem context respectively is advantageous to yield consistent results. Gauging requirements work packages to a comparable level of detail is in the focus of investigations in the following.

DEFINITION 5.4 Amount of Functionality

is all the information that make up the functional scope of a requirements work package. It

- determines the size of problems, and
- depends on a requirements decomposition that frames a problem context in a defined way.

For problem frames, the amount of functionality relates to the number of involved problem domains and their respective shared phenomena. For elementary processes, the amount of functionality relates to the number of involved base functional components and their respective data or control information, i.e. data element types, see also table 5.1 on page 48 and figure 5.5 on page 47.

In order to guarantee for equally sized problems, namely those which are of a comparable degree of requirements decomposition, requirements work packages require to build on functional size measurement patterns, that comprise a comparable functional scope for a unit of requirements. The following sections investigate proper intra- and inter-problem decomposition of requirements, respectively, for defining functional size measurement patterns that are of practical use. The hierarchical levels of detail as introduced in the following table 5.3 illustrate how much context specific to the measurement problem can be covered by a functional size measurement pattern, and it categorizes elementary processes and problem frames respectively.

[3]amongst other aspects, such as quality attributes, etc., which are not in the scope of considerations here

5.5.1. Hierarchical Levels of Detail

In its left column, table 5.3 makes use of a four-level pattern hierarchy for functional size measurement (FSM) patterns as developed by Trudel et al. [212], cf. section 5.2.4 Categories of Functional Size Measurement Patterns. The concepts of elementary processes as given by Trudel et al. as well as ISO 20926 in [117], and the problem frames as defined by Jackson [128] are arranged in this hierarchy, cf. section 5.2 for a brief Background on these. As a result of this arrangement, table 5.3 provides criteria in its columns two and three that allow to distinguish the respective levels. Consequently, it enables the identification of measurable problems, which share a comparable degree of requirements decomposition.

Hierarchy of FSM Patterns	Elementary Processes	Problem Frames	
I. Micro	· fragment of a transactional function · data functions are considered in isolation	Frame · no referenced domains	↑ limited
II. **Basic**	· self-contained yet **single** transactional fct. · data functions are considered in a unit(y) to one transactional function	**Basic Problem Frame** · at least one referenced domain · **exactly one** constrained domain	
III. Composite	· set of transactional functions that belong to **level II. Basic** · each features a **similar type** of functionality · · · relation of data functions involved is made obvious	Composite Problem Frame · sharing one constrained domain · assembled by several frames · joined in **one diagram**	**Functional Scope**
	· · · relation of data functions involved is not directly obvious	· joined by life-cycle-expressions for **separate diagrams**	
IV. Multi-Composite	· set of transactional functions that belong to level II. Basic · each features **different type**s of functionality · transactional functions are considered in relation to each other · · · relation of data functions involved is not in focus	Frame · more than one constrained domain · assembled by several frames · joined in one diagram possible	↓ broad

TABLE 5.3 Hierarchical levels of detail for functional size measurement patterns

The next paragraphs starting from page 56ff explain this hierarchy in detail.

An exemplary classification of some requirements, which demand a “write of a log file” (FUR wolf), illustrate the use and meaning of each level (I. Micro to IV. Multi-Composite) to the functional scope and respective size of a problem. Each of the four subsequent examples discuss the same requirement (FUR wolf) with increasing levels of problem context, e.g. different functional scopes or degree of detail, which are classifyable according to table 5.3 above.

Accordingly, section 5.6 Problem Pattern – Frames Revisited builds on this hierarchy to chose those patterns out of a set of problem frames, which fit to the level II. of basic functional size measurement patterns as covered by the gray-shaded area in table 5.3 above. This level and its respective patterns allow to establish sets of just – small and independent – enough software requirements. It enables the requirements analyst and estimator to build a requirements work package, which is comparable to others in regard to the problem to be solved and its functional size measurement.

Level I. micro problems

As illustrated by table 5.3 on page 55 in its first row, functional size measurement patterns at this hierarchical level of detail serve to classify requirements that describe a "**fragment** of an [...elementary] process" according to Trudel et al. [212].

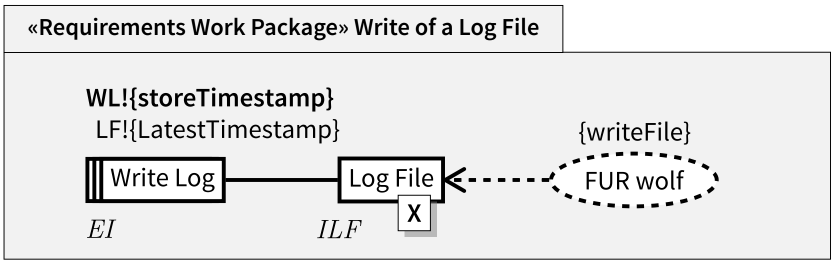
An elementary process belongs to one type of functionality as introduced in section 5.4.3 by table 5.2, and thus relates to one transactional function (EI, EQ, EO) and at least one additional data function (ILF, EIF), cf. also the meta-model in table 5.5 on page 47. So, it usually requires several base functional components out of ILF, EIF, EI, EQ, and EO to make an elementary process. Together, these form a unity of transactional and data functions, which jointly process information that crosses the application boundary.

For instance, receiving data from another software, which is seen as an EIF by ISO 20926, and processing this received data in a data base, which represents an EI operation (of TOFF-i.) on an ILF in ISO 20926, makes such a unit(y) of base functional components that finally consititutes one elementary process.

For the level I of micro problems, considering data functions in isolation is taken as characteristic for what a fragment of an elementary process is in functional size measurement.

In table 5.3, this fragment of a transactional function is put on a level with problem frames, that provide little context to the measurement problem, i.e. that have no referenced problem domains. This proposal for a mapping of elementary processes and problem frames to the level I of micro problems makes sense, since problems without referenced domains are usually integrated as part to other problems, that have referenced domains.

EXAMPLE 5.1 Level I. micro problem – write of a log file



For example, a requirement that demands a "write of a log file" (FUR wolf) can be classified as micro problem, which fits to level I of the FSM pattern hierarchy in table 5.3. It represents an instance of a *simple transformation* problem class namely **PF1.4** according to table 5.4 on page 70, that consists of only one constrained problem domain, which is in this example of a lexical type (**X**). FUR wolf is satisfied by the machine domain *Write Log*, which stores a timestamp to the *Log File*, such that it always contains the *lastest timestamp*.
From the perspective of functional size measurement, this problem compares to the consideration of a data function *Log File*, which is of type Internal Logical File (*ILF*), and which is processed by a transactional function that represents an External Input (*EI*), see also the meta-model in figure 5.5. By chosing the problem class of *simple transformation*, it remains unclear, in which problem context this data function is used, since only a part or excerpt of an elementary process is represented in this requirements work package. It shows only the data-related aspect of the measurement problem, but not its circumstances or trigger. In this case, there is not even a crossing of information along the application boundary, since an ILF remains inside the software under consideration, but only interaction and information between the software and its environment is of relevance to functional size measurement. That is why FUR wolf represents only a fragment of an elementary process in this example.

Functional size measurement patterns at the micro level I represent valid problem classes from the perspective of problem frames, but patterns at this level are too limited in their functional scope in order to provide for stand-alone problems that are of relevance for measuring purposes.

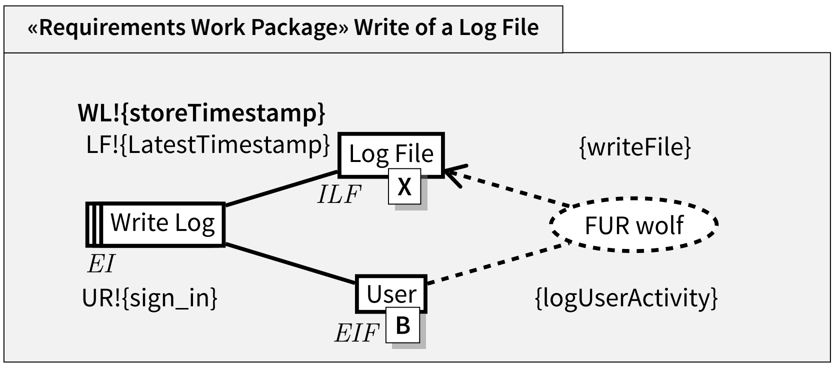
Level II. basic problems

As illustrated by table 5.3 on page 55 in its second row, functional size measurement patterns at this hierarchical level of detail serve to classify requirements that describe "a complete, yet **single** [...elementary] process" according to Trudel et al. [212].

Considering meaningful combinations of base functional components, i.e. data functions (ILF, EIF) in the context of a transactional function (EI, EQ, EO), at level II. of basic problems, conforms best to the definition of an elementary process in functional size measurement. It is defined as "smallest unit of activity that is meaningful to the user" [117, page 4, section 3.21], and it "must be self-contained and leave the business of the application being counted in a consistent state." [118] by ISO 20926.

In table 5.3, this self-contained, yet single process is put on a level with problem frames, that represent a basic problem "with the smallest number of domains" [128, page 361], i.e. one that has exactly one constrained and at least one referenced problem domain.

EXAMPLE 5.2 Level II. basic problem – write of a log file



For example, a requirement that demands a "write of a log file" (FUR wolf) can be classified as basic problem, which fits to level II of the FSM pattern hierarchy given in table 5.3. This requirement represents an instance of the *simple workpieces* problem class namely **PF2.7** according to table 5.4, that consists of only one constrained problem domain *Log File*, which is of a lexical type (**X**), and one referenced, biddable (**B**) domain *User*. FUR wolf is satisfied by the machine domain *Write Log*, which stores a timestamp to the *Log File*, such that it always contains the *lastest timestamp*. This machine control is invoked by a *User* as part of a sign-on procedure via initiating the *sign_in*.
From the perspective of functional size measurement, this problem compares to the consideration of a data function *Log File* which is processed by a transactional function of type external input (*EI*), and which in turn acts on specific conditions of the problem context, here on the *User*'s *sign_in*. Compared to the example for the level I of micro problems on page 56, the problem class or respectively the kind of functionality for FUR wolf remains the same. Both examples differing in the fact, that this time more problem context than only the data fragment, i.e. the *Log File* is shown. In this example, the functional scope of FUR wolf is now expanded to a level of detail, which reveals the *User*, who takes the role of an external interface file (*EIF*), as the trigger for this problem situation. In addition, all aspects of the measurement problem, the data (*ILF* and *EIF*) and the related transactional aspect (*EI*) which represent FUR wolf, are shown here in one meaningful combination, that is covered by a defined problem pattern, and thus representable by a requirements work package.

Functional size measurement patterns at the basic level II. represent the preferred mode of requirements decomposition for the sake of size measurement, and are thus objects worth of further investigations by this dissertation. At this level, problem frames and elementary processes establish units for requirements which share a comparable understanding of what makes a meaningful functional scope.

That is why the second row in table 5.3, namely level II. basic problems is highlighted in gray. It represents the level of detail for requirements classification and measurement for which functional size measurement patterns and a corresponding measurement method are designed in the following.

A subset of Jackson's composite problem frames does also belong to level II. basic problems, which is the reason for highlighting these in gray in table 5.3, too. Details on this fact are presented in the subsequent discussion of composite problems at level III.

Level III. composite problems

As illustrated by table 5.3 on page 55 in its third row, functional size measurement patterns at this hierarchical level of detail serve to classify requirements that describe "a **set** of [...self-contained, yet single processes] Instead of being restricted to a single [...one. These processes have] a high level functional meaning together [...and they] have the characteristic of sharing the same primary [...purpose]" according to Trudel et al. [212].

That is, data functions involved with this set of equally typed transactional functions are brought into one, shared relation. Means and reasons that make up this relation must be made explicit, in order to keep track of the number of processes that belong to a requirements set. Otherwise, functional size measurement would be futile, since it could not result in a comparable, and consistent number of function points.

In table 5.3, level III of the FSM pattern hierarchy relates to problem frames, that provide an extended context for a single measurement problem, i.e. it comprises those frames, which have exactly one constrained and several referenced problem domains, and in addition can represent a join of many problems into one (diagram) that share the same type of functionality. Section 5.5.2 Tailoring Measurable Problems – Pack the package provides further insights and approaches to this topic. The following example on page 60 gives more detail on what it looks like, when several elementary processes of the same type of functionality are joined into one diagramm or requirements work package, respectively.

In contrast to level II of basic problems, which consider single processes only, units for requirements at level III share the same type of functionality for several transactional functions, which of course "have overlapping scopes" [128, page 307] in regard to their data functions. On the one hand, these overlaps allow for a reasonable composition of requirements, but on the other hand, they are also a source of inconsistencies, particularily with regard to the problem size, if these overlaps are not revealed.

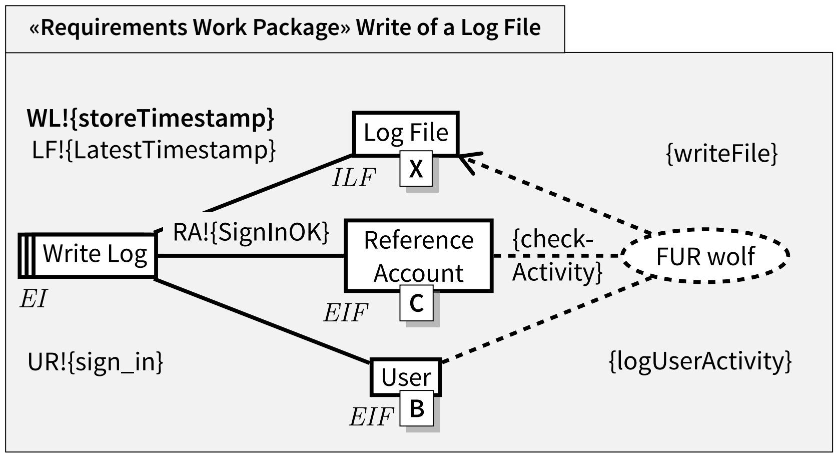
The relation of measurable problems that have overlapping functional scopes can be addressed in two, distinct ways, which yield different degrees of requirements decomposition. Either by the use of a *composite problem (frame)*, or by the use of a *composition of problem (frame)s*.

The first joins elementary processes by means of one frame diagram to a common instance of a measurable problem. This use and understanding of a composite problem frame is exemplified in the following on page 60. A composite problem frame at level III shares the same degree of requirements decomposition as problem frames of level II for basic problems. That is why the first row for composite problem frames in table 5.3 is also highlighted in gray and object to further investigations.

The second approach to address overlapping functional scopes joins elementary processes by means of life-cycle-expressions for composing several frame instances into one (measurable) problem. The use and understanding of a composition of problem frames is exemplified on page 62. Such a composition of problem frames represents a degree of requirements decomposition that is characteristic for FSM patterns at level III. The functional scope of this problem level is too broad for considering it within functional size measurement. Due to this reason, a composition of frame instances i.e. by life-cycle-expressions is not viewed as belonging to level II basic problems, and thus not included in the gray-highlighted cells of table 5.3. Nethertheless, knowing level III of composite problems, becomes of importance to manage requirements and related work package dependencies, which is a critical issue and contributes to project planning as is elaborated in part III of this dissertation.

The challenge is to make clear that the *level III. composite problems* as known from the hierarchy for functional size measurement patterns is not the same as *composite problem frames* as used by Jackson, even if the same wording may imply an equivalence. In the hierarchical levels of detail as discussed in this section, a subset of Jackson's composite problems frames belongs to level II of basic problems and another subset belongs to level III of composite problems, where only those at level II are of use in functional size measurement. The following examples apply the respective definitions given in table 5.3 to make the important differences clear.

EXAMPLE 5.3 Level III. composite problem – write of a log file



For example, a requirement that demands a "write of a log file" (FUR wolf) can be classified by a *composite problem (frame)*, such as the *commanded model building* problem class, see also **PF3.11** in table 5.4, which is a composite of *simple workpieces* with a *model building* problem into one common diagram. Their overlapping scopes relate to the storing of the timestamp in the constrained, lexical (**X**) *Log File* problem domain. This composite problem constitutes one measurable problem, which belongs to level II basic problems of the functional size measurement pattern hierarchy given in table 5.3 (highlighted in gray), and which is of use by requirements work packages as defined on page 46. FUR wolf is satisfied by the machine domain *Write Log*, which stores a timestamp to the *Log File*, such that the log file always contains the *latest timestamp*. This machine control is invoked by a *User* as part of a (single-)sign-on procedure, and accompanied by a credentials check at a *Reference Account*, which is an other software application that holds own data[a].
From the perspective of functional size measurement, the transactional function *Write Log* can be regarded as a composition of two elementary processes, which share one type of functionality (TOFF-i. see table 5.2). Both, the *simple workpieces* as well as *model building* problem (processes) are instances of an external input (*EI*). Their particular, unique data functions, namely *User* and *Reference Acount*, are shown in one meaningful unit(y) as a measurable problem. The choice of the *commanded model building* problem class establishes a requirements work package, which takes their overlapping requirements constraint on the same problem domain *Log File* into account. Compared to the level I of micro problems on page 56, and the level II of basic problems on page 57, the problem class for the example to level III of composite functional size measurement problems remains the same again, namely an external input (*EI*). Only the number of data functions and thus the scope of the problem context evolves.

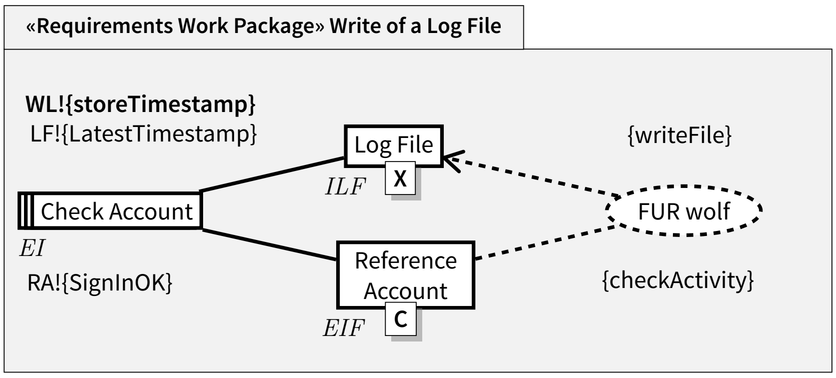
[a]Note: In this example, *Reference Account* is another software application classified as a causal (**C**) problem domain, which takes the role of an external interface file (*EIF*, cf. figure 5.3 on page 43 and its respective explanation) to the software machine under consideration, here the *Write Log*. *Reference Account* is in control of the symbolic phenomenon *SignInOK*, which it makes available to *Write Log* for further processing. As introduced in section 4.3, this work focuses on developing software to build information systems. That is why causal domains are used to mark software application (part)s other than the one(s) to be built. The importance and use of this is detailed further in respective sections.

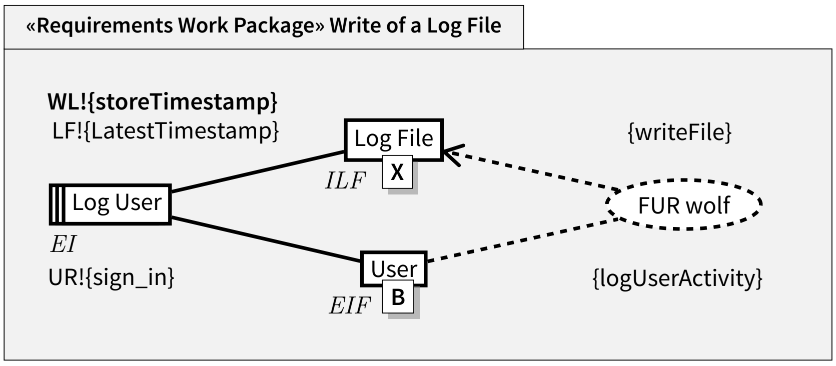
Functional size measurement patterns at the composite level III that relate a set of elementary processes by one common composite problem frame such as given in the example above equal in their functional scope to patterns of basic problems at level II of the functional size measurement hierarchy given in table 5.3.

Thus, even if level III is about composite FSM patterns, composite problem frames as known from Jackson [128] have only the name in common with this level, but they do not coincide with the respective functional scope at this level of requirements decomposition. Jackson's composite frames do not suffice to make the composition of elementary processes (or the respective existence of a set of elementary processes by one diagram) explicit as is needed for determining reproducible and thus comparable function point counts.

In contrast to level I and level II of the FSM pattern hierarchy and as a result found by the example above, functional size measurement patterns at the composite level III are not representable by one defined frame diagram. In order to be in the position to represent measurable problems and to classify their respective functional scope to level III of the pattern hierarchy, life-cycle-expressions are applied to reveal the problem composition as is shown next.

EXAMPLE 5.4 Level III. composite problem, continued – write of a log file





problem composition of **FUR wolf ::= {CheckAccount || LogUser}**

For example, a requirement that demands a “write of a log file” (FUR wolf) can fit to a *composition of problem (frame)s* as mentioned on page 58 to build a level III composite problem as defined by the pattern hierarchy given in table 5.3. For instance, the consideration of FUR wolf at level III yields a **set of** two elementary processes or respective requirements work packages, of which each features a **similar type of functionality** (EI). Both requirements work packages have a requirements constraint on the same problem domain *Log File* in common.
The upper requirements work package for *Check Account* is an instance of a *model building* problem pattern, which considers FUR wolf in a different context than the lower requirements work package for *Log User*, which is an instance of a *simple workpieces* frame. FUR wolf is satisfied by a (parallel) composition of both, represented as FUR wolf ::= {CheckAccount || LogUser}.
From the perspective of functional size measurement, it can be said, that each of these problems represents the same elementary process, which is integrated to different aspects (in one case to the *Reference Account* in the other to the *User*) of a common problem scope (both overlap in regard to the storage of a timestamp in *Log File*). In other words, both requirements work packages represent variations of finally the same functional size measurement problem, which is why it is reasonable to consider these in combination.

At level III of the FSM pattern hierarchy, life-cycle expressions are applied to glue measurable problems together, which feature similar types of functionality in regard to a common requirements constraint.

It remains to decide, if *Check Account* and *Log User* represent one measurable problem as illustrated on page 60, or if these constitute two requirements work packages as given here. Since this decision impacts their functional size, this issue is elaborated in detail in section 5.5.2 Tailoring Measurable Problems – Pack the package.

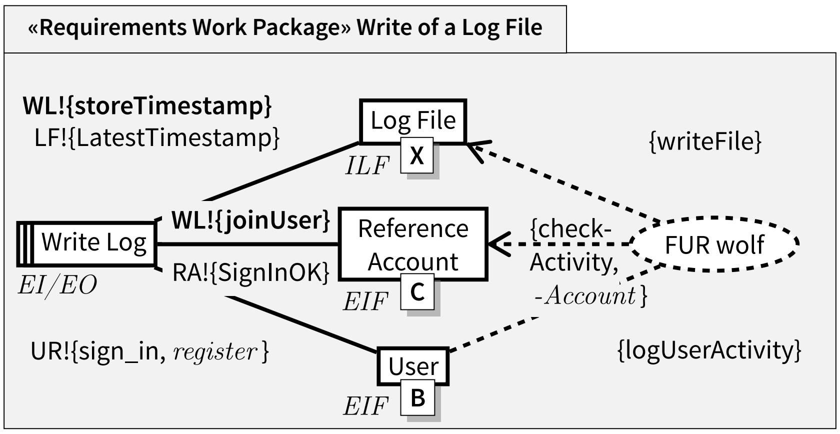
In summary, composite functional size measurement problems at level III are described by means of life-cycle-expressions that compose level II basic problems. There are no functional size measurement patterns or problem frames, which represent this level of the FSM pattern hierarchy by themselves. That way, formerly tacit problem relations become revealed, which serves the (de)composition of requirements (to) work packages and their consistent size estimation.

Level IV. multi-composite problems

As illustrated by table 5.3 on page 55 in its fourth row, functional size measurement patterns at this hierarchical level of detail serve to classify requirements that describe “a set of composite and [...and basic problems] having functional relationships among them. [...It] combines multiple [...] processes handling several [...purposes]” [212].

In table 5.3, this set of multiple processes is put on a level with problem frames, that represent a composition of basic problems that feature different types of functionality, i.e. those that have more than one constrained and several, referenced problem domains.

EXAMPLE 5.5 Level IV. multi-composite problem – write of a log file



For example, a requirement that demands a “write of a log file” (FUR wolf) can be classified as a multi-composite problem, which fits to level IV of the pattern hierarchy.
It represents an instance of an *update* problem class according to [49], which is a composite of a *simple workpieces* with a *commanded behavior* problem. The two constrained problem domains *Log File* and *Reference Acount* indicate a composition of processes that belong to different types of functionality. In this case, the *simple workpieces* process (part of the machine) is of a TOFF-i. according to table 5.2, which relates to an external input (EI) in ISO 20926, and the *commanded behavior* process is of a ToFF-iii., which relates to an external output (EO) in functional size measurement according to ISO 20926.
From the perspective of functional size measurement, this update of a log file on a user *sign_in* represents two measurable problems at once that require individual consideration.

Functional size measurement patterns at the multi-composite level IV represent a degree of requirements decomposition, which are supported by some problem frames like the *update* one, but they represent problems, which are of no proper functional scope for the sake of measuring their functional size by ISO 20926 [117].

Unique problems require unique measurement. Otherwise, consistency of measurements cannot be guaranteed. That is, only processes that are concerned with the same type of functionality need to be jointly considered in one requirements work package. The composite level III of the pattern hierarchy offers by means of basic problem patterns and life-cycle-expressions an alternative way to cope with multi-composite problems.

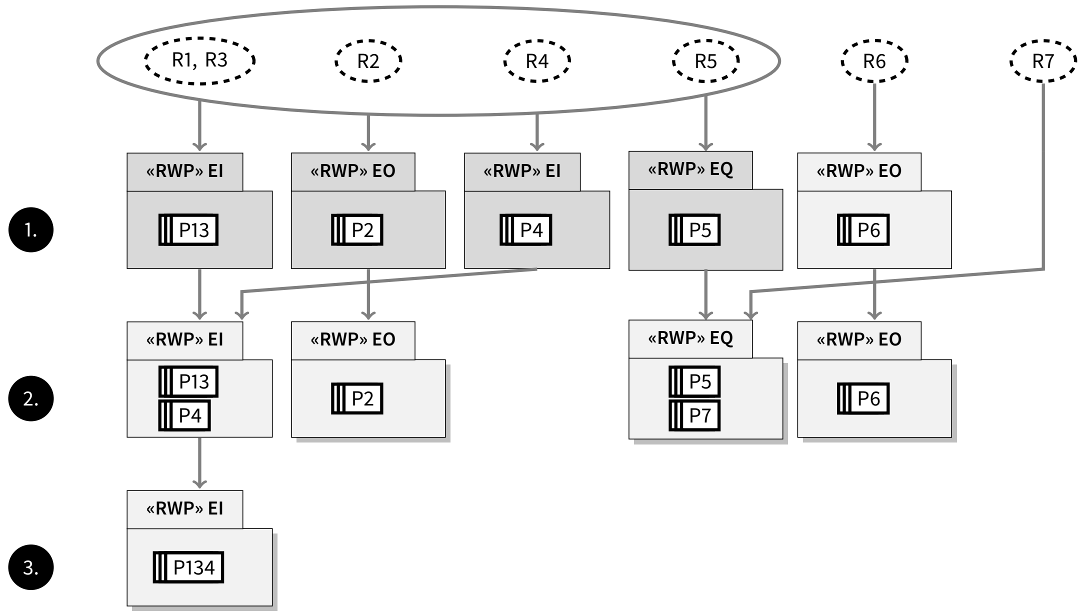
Finally, only patterns at the level II basic problems represent a level of detail and respective requirements decomposition, which is reasonably applicable for functional size measurement. The patterns in table 5.5 are chosen accordingly. These ensure the development of self-contained and measurable requirements work packages as demanded by section 5.3 using patterns, which enable reproducible requirement estimates.

5.5.2. Tailoring Measurable Problems – Pack the package

In order to care for consistent requirement estimates, it is not solely important to decide "what *can* be counted and what *cannot* be counted we always have to determine whether the 'variation' is a separate and unique *functional* requirement of the user." [209, page 1], which is assisted by patterns as developed in this dissertation. It is also mandatory to mitigate the risk of double counted problems, and thus of accidently inducing wrong numbers of function points for requirements to their subsequent prioritization and planning in a software project.

Hitherto exisiting approaches rely on the use of criteria such as given by Total Metrics [209] and IFPUG [117, page 14, section 5.5.2.2] to clarify for each emerging requirement, if it belongs to an already exisiting requirements set. That is, these criteria allow to set up groups, families or any other arbitrarily chosen form of work packages for requirements in relation to their underlying functional process. This manual check can not only become a very exhaustive one, but also a defective procedure in itself.

By use of problem frames that belong to the level of basic functional size measurement patterns as in table 5.5 on page 72, this situation becomes much more comfortable, since these problem-based functional size measurement patterns provide for a structuring, which deliberately constrains the class and scope of requirements to that of a single elementary process. This is very beneficial for creating requirements work packages that implement a coherent set of software functionality with regard to their size.



Legend:

R1 to R7	requirement statements
«RWP» {EI|EQ|EO}	requirements work packages implementing a particular elementary process
P1 . . . 7	machine icons for representing (joint) measurable problems

FIGURE 5.6 Cases for unique functionality within requirements work packages

Nevertheless, instantiating several, but differently framed instances of finally the same measurable problem must be avoided for ensuring consistent requirement estimates. As introduced by Example **EXAMPLE 5.4** on page 62, for each measurable problem it must be checked, if it is a unique, stand-alone one or a variant of an other, already identified, measurable problem, and thus capable of being integrated to a joint requirements work package.

Figure 5.6 illustrates several cases of how to establish those requirements units, which each for themselves exhibit unique software functionality. These units are identified and managed through a requirements decomposition, that takes requirements dependencies into systematic account.

As designed by this work, Definition **DEFINITION 5.5** in synergy with problem-based functional size measurement patterns as in table 5.5 on page 72, assists this requirements decomposition by framing units of unique software functionality, whose dependencies are made transparent. Knowing these units and their dependencies is a prerequisite to achieve consistent requirements estimates.

DEFINITION 5.5 Unique Set of Functionality (UF)

can be determined by the following criteria **UF.C1** and **UF.C2**. Accordingly, measurable problems involved with two RWPs are joined into one RWP and counted as the same,

UF.C1 if these fit to the same problem class (EI, EQ, EO)

- with respect to their requirements constraint for a shared problem domain. And

UF.C2 if these, in addition to **UF.C1**, have overlaps in their problem scope (DET)

- with respect to some of their shared phenomena,
- which do not change independently of each other[a].

  Problems within one RWP that fulfill **UF.C2** are combined into one new, measurable problem, which has merged respective overlapping phenomena.

[a]Note: If two problems are meaningful in a parallel composition, such as in Example **EXAMPLE 5.4** on page 62, their common phenomena provide an anchor point for combining these problems into one, as $WL!\{storeTimestamp\}$ and $LF!\{LatestTimestamp\}$ at the machine interface to *Log File* in both problems *CheckAccount* and *LogUser*. These anchor points or problem overlaps indicate why the other phenomena do not change independently of each other. In case of the chosen example, this means that the change to *Log File* depends on both $US!\{sign\_in\}$ and $RA!\{SignInOK\}$. The RWP in Example **EXAMPLE 5.3** on page 60 would be one possible resulting merger for this example.

An explanation of figure 5.6, which illustrates this pattern-supported requirements decomposition into measurable units of unique software functionality follows.

1. The initial situation is marked by a gray-colored ellipse, which consists of a set of requirement statements $R1$ to $R5$. These requirements are grouped by means of problem-based functional size measurement patterns into four requirements work packages, which represent the measurable problems $P13, P2, P4$ and $P5$. Each of these implements a particular elementary process, which can be an external input (EI), external inquiry (EQ), or external output (EO).

As illustrated here, executing requirements decomposition may in a worst case result in one individual requirements work package for each requirements statement, which comes naturally but is hardly likely desired. The unsurpassable advantage of using patterns, is the gained knowledge about the problem class, which speeds up subsequent requirements analysis and respective decomposition tremendously, as shown next.

In case new requirements emerge and these do not fulfill Definition **DEFINITION 5.5** criteria **UF.C1**, they result a new requirements work package, such as $R6$. For this example it is assumed that $P2$ and $P6$ place a requirements constraint on different problem domains, which prohibits their merger into one requirements work package, even if these are instances of the same type of functionality or elementary process EO. If a new requirement fulfills Definition **DEFINITION 5.5** criteria **UF.C1**, it is integrated to an exisiting requirements work package, such as in case of $R7$, which becomes integrated to the requirements work package *«RWP»* EQ of $P5$.

In any case, due to the structuring provided by the patterns, it becomes much easier to clarify requirements dependencies and to decide, if a supposed new problem is (part of) an already exisiting one.

2. Consequently, there are up to three cases of requirements decomposition or ways to form unique functionality, whose identification is enabled by problem-based functional size measurement patterns and which are managable through the

- criterion **UF.C1** given in Definition **DEFINITION 5.5**. This criterion allows for maintaining **inter**-work package dependencies of requirements. It enables the
  - Integration of a new requirement to an already exisiting requirements work package, e.g. $R7$ becomes part of *«RWP»* EQ for $P5$.
  - Integration of exisiting requirements work packages to one, e.g. the two stand-alone requirements work packages *«RWP»* EI for $P13$ and $P4$ become members of one RWP. Both share the same problem class and requirements constraint on the same problem domain as assumed in this example. This makes them instances of the same measurable problem. In contrast, the two *«RWP»* EO for $P2$ and for $P6$ remain as-is, since these are instances of different problems as is indicated by different, constrained problem domains.

  That way, criterion **UF.C1** ensures that each requirements work package implements a complete problem in itself.
- criterion **UF.C2** given in Definition **DEFINITION 5.5**. This criterion allows for maintaining **intra**-work package dependencies of requirements. It enables the
  - Merge of two problems within one requirements work package, e.g. as for $P13$ and $P4$, which become one $P134$.

  Due to this criterion **UF.C2**, it is ensured that each requirements work package implements just small enough problems.

3. Due to the fulfillment of criteria **UF.C2**, the requirements work package *«RWP»* EI for $P13$ and $P4$ must be further consolidated to care for consistency. The two problems must be merged into one problem as is illustrated in more detail by Example **EXAMPLE 5.4**, which also illustrates the importance of the criteria given in Definition **DEFINITION 5.5**. Finally, seven requirement statements $R1..R7$ are grouped into five measurable problems $P134, P2, P5, P7, P6$, which are represented by four requirement work packages only, each containing a unique set of cohesive software functionality.

Tailoring measurable problems in this way extents the use of problem patterns in early functional size measurement beyond the scope of requirement estimating. Such a packaging of requirement statements is not only beneficial for establishing consistent estimates but also for managing requirements complexity by making their composition and involved dependencies transparent. More details on this issue are presented in the Case Studies.

EXAMPLE 5.6 Determine a unique set of functionality – Pack the package for FUR wolf

Example **EXAMPLE 5.4** on page 62 presents two stand-alone requirements work packages named $Check\ Account$ and $Log\ User$, which constitute one composite problem that is maintained by means of a life-cycle expression FUR wolf ::= {CheckAccount || LogUser}.
It leaves the decision open, if joining these two requirements work packages into one is desirable, such that a requirements work package as in Example **EXAMPLE 5.3** on page 60 is obtained, which would make maintaining an additional life-cycle expression, and thus a potential source of inconsistencies obsolete. That is why the criteria for determining a unique set of functionality as given by the Definition **DEFINITION 5.5** on page 66 are used to resolve this issue.
This example conforms to the integration and merger of $P13$ and $P4$ in figure 5.6.

☑ **Criterion UF.C1 The two RWPs** $Check\ Account$ **and** $Log\ User$ **fit to the same problem class with respect to a common constrained problem domain (EI,EQ,EO).**

Even though both RWPs are instances of different problem patterns, namely $model\ building$ and $simple\ workpieces$ as given by table 5.5, these represent the same problem class, which is an external input (EI) or TOFF-i. according to table 5.2. Both RWPs share a requirement constraint on a common problem domain, which is placed on $Log\ File$.

That way, criterion **UF.C1** implies that both RWPs represent at least variants of finally the same measurable problem and thus must become members of one requirements work package.

Compared to figure 5.6, this conforms to the integration of $P13$ and $P4$ to one requirements work package.

☑ **Criterion UF.C2 The two RWPs** $Check\ Account$ **and** $Log\ User$ **have overlaps in their problem scope.**

static overlap: **Both RWPs have some shared phenomena within their problem context in common.**

Shared phenomena common to different RWPs, which are in this example $LF!\{LatestTimestamp\}$ and $WL!\{StoreTimestamp\}$, represent a potential risk for double counts, which must be mitigated. That way, this first indication of criterion **UF.C2** further strengthens the need to join these two RWPs $Check\ Account$ and $Log\ User$ into one, such that $LF!\{LatestTimestamp\}$ and $WL!\{StoreTimestamp\}$ are only considered and counted once.

dynamic overlap: **Both RWPs have shared phenomena in common, which do not change independently from each other.**

This second indication of criterion **UF.C2** is confirmed by the parallel composition of these two RWPs in the life-cycle expression FUR wolf ::= {CheckAccount || LogUser}, which indicates that these are intertwined in such a way that they do not change the $Log\ File$ independently from each other[a]. In this context, the problem of $Log\ User$ can be understood as the triggering for or a problem that involves the functionality of $Check\ Account$. The multiple appearance of the shared phenomena $LF!\{LatestTimestamp\}$ and $WL!\{StoreTimestamp\}$ in both RWPs can be resolved by merging these into one combined problem, such as given by example Example **EXAMPLE 5.4**, without loss in expressiveness and for the advantage of consistent requirements estimates.

Compared to figure 5.6, this conforms to the merger of $P13$ and $P4$ to one combined problem $P134$.

[a]Note: Only the parallel dependence of requirements is of importance here. Any other relation, e.g. sequential or alternative, already implies that the requirements address different purposes even in the presence of problem overlaps.

5.6. Problem Pattern – Frames Revisited

As elaborated in the previous sections, problem frames and elementary processes have comparable conceptions to consider units of requirements. Both comprise a set of software functionality, which exhibits a specific type of functionality and level of detail. That way, self-contained units of requirements can be established and associated with a functional size.

This section revisits some former work [68], where all possible problem frames have been created by permutation and compiled to a comprehensive list, which is given in table 5.4 on page 70. It makes use of an alternative, short-cut representation of problem frames in a tabular form, instead of using frame diagrams. Based on the findings from the previous sections, this work proposes to choose problem frames that belong to the second, basic level II of the functional size measurement pattern hierarchy as given in table 5.3 on page 55 to set up measurable problems.

DEFINITION 5.6 Measurable Problem

is a unit of software functionality, which fits to a

- basic functional size measurement pattern.

Level II. of basic problems and respective functional size measurement patterns frame a self-contained unit of software functionality in a way that it comprises one, independent elementary process. In addition, these patterns can be used for integrating process fragments from the level I of micro problems and for expressing (multi-)composite processes, which are level III and level IV problems.

Table 5.4 on page 70 summarizes the frame permutations as given by Côté et al. [68, tables 2, 3, and 4]. Table coloring indicates changed annotations, and the right column **level** is an extension of the original publication, which are explained in the following.

The right column **level** gives the classification of each problem frame to its level of detail according to table 5.3 on page 55. There is no classification of problem frames to the composite or multi-composite level in this table, since only single frames are considered (no grouping of these), which hold only one constrained problem domain (no multiple requirement constraints per frame diagram), just as done in the respective publication [68].

PF 1.2 to **PF 1.4** are no Basic FSM patterns. They belong to the first level in the hierarchy representing micro problems, since they have no referenced problem domain. Requirements that fit to these problems should be combined with other, basic problems in order to maintain a consistent level of problem decomposition.

PF 2.1 is no Basic FSM pattern, since it does not represent a self-contained yet single elementary process, i.e. PF 2.1 is not an external input, see discussion in section 5.4.2. Accordingly, information received from and sent back to a lexical (X) domain resides within the application boundary. Since only data and control information that crosses the boundary counts in functional size measurement, problems that fit PF 2.1 should be addressed by another pattern or decomposed differently, cf. the previous section 5.5.2.

All other problem frames in table 5.4, which are no Basic FSM pattern, represent either an invalid frame permutation, which contradicts one of the integrity conditions, or the frame is interchangeable with an already considered frame permutation, or its domains can be merged according to the rules given below, which results again a problem frame, that is already a member of the permutation list.

The following section 5.6.1 Integrity Conditions and section 5.6.2 Merge Rules summarize changes made to the integrity conditions and merge rules as applied to table 5.4 in this work. This is important, since integrity conditions and merge rules define the validity of a problem frame. Section 5.6.3 Problem-Based Functional Size Measurement Patterns presents and discusses the final choice of frames, which are applicable as problem-based functional size measurement patterns.

Problem Frame No.	constrained problem domain	1st referenced domain	2nd referenced domain	comment	**Level, acc. to tab. 5.3**
PF 1.1	B	-	-	n/a: a biddable domain cannot be constrained (↯IC02)	n/a
PF 1.2	C	-	-	Jackson [128, page 85]: required behaviour	I. Micro
PF 1.3	D	-	-	new: generated information	I. Micro
PF 1.4	X	-	-	new: simple transformation	I. Micro
PF 2.1	X	X	-	Jackson [128, page 99]: transformation	I. Micro
PF 2.2	C	X	-	new: data-based control	**II. Basic**
PF 2.3	D	X	-	Jackson [128, page 194]: model display	**II. Basic**
PF 2.4	X	C	-	Jackson [128, page 198]: model building	**II. Basic**
PF 2.5	C	C	-	**NOT** merged frame: PF 1.2, (MR***) → **new**: required behavior (variant)	**II. Basic**
PF 2.6	D	C	-	Jackson [128, page 93]: information display	**II. Basic**
PF 2.7	X	B	-	Jackson [128, page 96]: simple workpieces	**II. Basic**
PF 2.8	C	B	-	Jackson [128, page 89]: commanded behaviour	**II. Basic**
PF 2.9	D	B	-	new: commanded display	**II. Basic**
PF 2.10	B	⋆	-	n/a: a biddable domain cannot be constrained (↯IC02)	n/a
PF 2.11	⋆	D	-	n/a: a display domain must be constrained (↯IC01)	n/a
PF 3.1	X	X	X	merged frame: PF 2.1	cf. PF 2.1
PF 3.2	X	X	C	**NOT** merged frame PF 2.4 → **new**: triggered transformation	**II. Basic**
PF 3.3	X	X	B	Wentzlaff&Specker [224, 228]: commanded transformation	**II. Basic**
PF 3.4	⋆	⋆	D	n/a: a display domain must be constrained (↯IC01)	n/a
PF 3.5	C	X	X	merged frame: PF 2.2	cf. **PF 2.2**
PF 3.6	C	X	C	interchangeable with PF 3.12	cf. **PF 3.12**
PF 3.7	C	X	B	new: commanded data-based control	**II. Basic**
PF 3.8	D	X	B	Choppy&Heisel [49]: query	**II. Basic**
PF 3.9	X	C	X	interchangeable with PF 3.2	cf. **PF 3.2**
PF 3.10	X	C	C	merged frame: PF 2.4	cf. **PF 2.4**
PF 3.11	X	C	B	new: commanded model building	**II. Basic**
PF 3.12	C	C	X	**NOT** merged frame: PF 2.2 → **new**: triggered data-based control	**II. Basic**
PF 3.13	C	C	C	merged frame: PF 1.2	cf. PF 1.2
PF 3.14	C	C	B	**NOT** merged frame: PF 2.8 → **new**: commanded behaviour (variant)	**II. Basic**
PF 3.15	D	C	B	Jackson [128, page 215]: commanded information	**II. Basic**
PF 3.16	X	B	X	interchangeable with PF 3.3	cf. **PF 3.3**
PF 3.17	X	B	C	interchangeable with PF 3.11	cf. **PF 3.11**
PF 3.18	X	B	B	**NOT** new: multi-user simple workpieces → merged frame: PF 2.7	cf. **PF 2.7**
PF 3.19	C	B	X	interchangeable with PF 3.7	cf. **PF 3.7**
PF 3.20	C	B	C	interchangeable with PF 3.14	cf. **PF 3.14**
PF 3.21	C	B	B	**NOT** new: multi-user commanded behavior → merged frame: PF 2.8	cf. **PF 2.8**
PF 3.22	B	⋆	⋆	n/a: a biddable domain cannot be constrained (↯IC02)	n/a
PF 3.23	⋆	D	⋆	n/a: a display domain must be constrained (↯IC01)	n/a
PF 3.24	D	C	X	**new**: triggered information	**II. Basic**

Color code:

light gray	Basic functional size measurement patterns according to table 5.3 Hierarchical levels of detail for functional size measurement patterns
dark gray	**new** frame or altered rationale compared to Côté et al. [68]

Legend:

Domain type:	Le(X)ical, (D)isplay, (C)ausal, (B)iddable
⋆	any arbitrary domain type possible
–	no domain at all

TABLE 5.4 Problem Frame permutations as published in [68], revisited

5.6.1. Integrity Conditions

Integrity Conditions **IC01** to **IC02** have been proposed by Côté et al. [68] to maintain consistency and validity of considered frames in table 5.4 on page 70. They are not changed in this work.

IC01 A domain of type display is always constrained in a problem frame.

IC02 A biddable domain is never constrained in a problem frame.

5.6.2. Merge Rules

Merge rules **MR01** to **MR03** have been proposed by Côté et al. [68]. They are revised and entirely substituted by **MR***** in this work to set up table 5.4 on page 70 due to the following reasons: **MR03** and **MR02** argue at the level of problem diagrams (instances) and not at the level of frames (patterns). **MR01** ignores the difference between a requirements reference and a requirements constraint on a problem domain, which justifies its alteration to **MR*****. All merged problem frames in table 5.4 Problem Frame permutations as published in [68], revisited have been reworked according to this new merge rule.

MR*** Two domains of the same type can be merged,
if they are not referenced and constrained at the same time.

MR01 Two domains can be merged, if they share the same domain type.

MR02 Two domains can be merged, if a good name representing both domains can be found.

MR03 Two domains can be merged, if one domain is related to the other domain by aggregation.

5.6.3. Problem-Based Functional Size Measurement Patterns

In the previous section, table 5.4 identifies a choice of frames as basic functional size measurement patterns, which are all capable to describe the information involved with a measurable problem at a comparable level of detail (cf. problem scope in section 5.5).

In this section, the review of table 5.4 is resumed by consolidating its choice of frames with regard to their type of functionality (cf. problem class in section 5.4). It determines the rules that apply for measuring the functional size of a problem that is represented by some requirements.

In this way, the revised set of patterns given in table 5.5 are applicable to scale the problem description for a requirements set, such that each maintains the same level of decomposition, and each is measured according to the same rules.

By comparison with the size and complexity tables used in functional size measurement, which are given in the appendix A on page 262, it becomes clear that the number of data element types (DET) and knowledge about the base functional components (ILF, EIF, EI, EO, EQ), which coincides with the amount and kind of functionality addressed by a set of requirements, are the key to a requirements' size measure. The meta-model Conceptualization of a Requirements Work Package developed in figure 5.5 on page 47 meets this concern.

Thus, problem-based functional size measurement patterns come up with built-in means to control these two most critical parameter to a measurable problem's functional size. Each pattern in table 5.5 is a smart tool to reproducibly set up requirements work packages (cf. problem unit in section 5.3), which comprises measurable problems that are comparable in their functional sizes.

The following explanations motivate the final set of frame permutations in table 5.5 that is considered as sufficient for problem-based functional size measurement.

	Problem Frame	Referenced	Referenced	Constrained	Basic Problem	Elementary Process		
#	No.	Domain I	Domain II	Domain ⇠	Frame Name	EI ←	EQ ⇝	EO →
01	**PF 2.4**		C	X	model building	TOFF-i.		
02	**PF 2.7**		B	X	simple workpieces	TOFF-i.		
03	**PF 3.2**	C	X	X	triggered transformation	TOFF-i.		
04	**PF 3.3**	B	X	X	commanded transformation	TOFF-i.		
05	**PF 3.11**	B	C	X	commanded model building	TOFF-i.		
06	**PF 2.3**		X	D	model display		TOFF-ii.	
07	**PF 2.6**		C	D	information display		TOFF-ii.	
08	**PF 2.9**		B	D	commanded display		TOFF-ii.	
09	**PF 3.8**	B	X	D	query		TOFF-ii.	
10	**PF 3.15**	B	C	D	commanded information		TOFF-ii.	
11	**PF 3.24**	C	X	D	triggered information		TOFF-ii.	
12	**PF 2.2**		X	C	data-based control			TOFF-iii.
13	**PF 2.5**		C	C	required behavior (variant)			TOFF-iii.
14	**PF 2.8**		B	C	commanded behavior			TOFF-iii.
15	**PF 3.7**	B	X	C	commanded data-based control			TOFF-iii.
16	**PF 3.12**	X	C	C	triggered data-based control			TOFF-iii.
17	**PF 3.14**	B	C	C	commanded behavior (variant)			TOFF-iii.

Legend:

Elementary process:	External Input (EI), External Inquiry (EQ), External Output (EO)
Domain type:	Le(X)ical, (D)isplay, (C)ausal, (B)iddable
TOFF-i. to TOFF-iii.:	Problem frame relates to elementary process with regard to their jointly considered type of functionality (cf. table 5.2 Mapping problem frames to elementary processes by types of functionality)

TABLE 5.5 Basic Problem Frames with relevance in Functional Size Measurement

In contrast to Côté et al. [68], several frame permutations in table 5.4 are not merged due to a new merge rule that applies: **MR*****, see section 5.6.2. Therefore, a new frame is created for **PF 2.5** named: Required Behavior (variant). The **PF 3.2** becomes a new frame named: Triggered Transformation, **PF 3.12** becomes a new frame named: Triggered Data-based Control, **PF 3.14** becomes a new frame named: Commanded Behavior (variant), and **PF 3.24** is newly formed in this work, it becomes a new frame named: Triggered Information. **PF 3.18** and **PF 3.21** can be ignored, since they are covered by PF 2.7 and PF 2.8.

As a result, table 5.5 presents 17 problem frames that allow for classifying requirements to basic measurable problems. Within requirements engineering this set of patterns represent reusable units of measure for requirements, which serve to set up comparable units of software functionality and enable to determine the functional size of a software development problem in a reproducible way.

Basic Problem Frames with relevance in Functional Size Measurement are fundamental to the frame counting procedure, which implements a customized functional size measurement process according to ISO/IEC 20926:2009 and that is developed and discussed in chapter 6 Problem-Based Estimating Method.

5.7. Discussion & Related Work

Problem-based units of measure as developed in this chapter are in first instance an instrument for scoping software product requirements in a defined way, which is to the advantage of consistent requirement estimates. This is made possible by means of patterns, that account for a reproducible requirements decomposition and which explicitly utilizes requirements dependencies.

Therefore, problem-based functional size measurement patterns serve as reusable problem templates for structuring requirements, which allow to fill a coherent set of desired software functionality in equally sized problem containers. These problem containers or problem-based units for requirements definition and measurement are designed and referred to as requirements work packages in here.

Consistent requirement estimates demand the identifiability of comparable requirements that require "a similar amount of work" [58] according to Cohn, and to join these into groups that are meaningful for estimating to most members of the project team, as noted by Daneva et al. in [73, page 1343]. Palomares et al. (2016) suggest, that this is preferably be done by means of "requirements patterns as a particular strategy to reuse" [169, pages 1 – 2], which provides the project team with a common point of reference regarding the recognition of recurring classes of measurable problems. According to Wiegers and Beatty [226, chapter 18], this form of requirements reuse also reveals their shared understanding of the requirements and it also provides an opportunity to identify emerging as well as applicable best practices.

Implementing a pattern-guided strategy to requirements reuse as a basis for determining consistent point values for these is addressed in this work by combining problem-oriented software engineering and in particular its problem patterns with the concepts used for functional size measurement. Such a combination is presented for the first time by Lavazza and del Bianco [141] in 2008, where an "initial investigation" on the general applicability of problem frames to functional size measurement appears to be promising. That work is resumed by them [32], where problem frames are used to set up UML sequence diagrams for functional user requirements as a means that is "fairly easy to measure" following the COSMIC counting regime for functional size measurement. Already in 1999, Uemura et al. [215, figures 3 – 7] described five patterns, that indicate measurable components in sequence diagrams of UML design specifications. Making use of design patterns and components as a means for estimating software size is illustrated by Troche (2004) using a sample application. None of these available works make use of requirement patterns as the first class means to operate the functional size measurement process, which is one important concern of this dissertation.

In addition, integrating requirements pattern with functional size measurement is not only advantageous for consistent problem counting, i.e. requirements estimating, but also for maintaining a consistent problem composition. This is of high relevance to subsequent prioritization and planning of requirements in a project, since their number und involved dependencies (will) scale up. Therefore, it is argued by this dissertation to enforce control on the requirement's complexity by tailoring the problem scope to a basic (level II., see table 5.3) degree of requirements decomposition.

A respective choice of problem frames in table 5.5 on page 72, and criteria on page 66 to set up sets of unique software functionality are provided, that allow to systematically address requirements dependencies and thereby eases to maintain the aspired degree of separation of concerns for each measurable problem. Working with this limited set of problem-based functional size measurement patterns is reasonable, since it allows for establishing highly flexible and expressive requirements work packages. These are capable of mounting (level I.) micro problems such that involved assumptions on their problem context, which are represent through problem domains, become apparent, and to assemble (level III. and level IV. multi-)composite problems by the use of life-cycle expressions. The effectiveness and sufficiency of this chosen set of problem frames is demonstrated in the Case Studies part.

5.8. Summary

This chapter elaborates on the design of seventeen problem-based functional size measurement patterns given in table 5.5 Basic Problem Frames with relevance in Functional Size Measurement on page 72, that are applicable for decomposing requirements into measurable problems. These make determining of consistent function point values for comparable requirements possible.

Each measurable problem is represented by a requirements work package. It unites requirements with special attention to their involved kind and amount of functionality, which are investigated in detail in section 5.4 Problem Class – Kind of Functionality and section 5.5 Problem Scope – Amount of Functionality. These are the two key parameters to a problem's functional size as defined by ISO/IEC 20926:2009 size and complexity tables A.1 to A.5 and given in the appendix A on page 262ff in this work. Problem-based functional size measurement patterns reconcile desired software functionality with these parameters. That is why each requirements work package constitutes a coherent set of desired software functionality that exhibits three built-in properties: first, it belongs to a measurable problem class, which relates to one out of three types of functionality as developed in table 5.2 Mapping problem frames to elementary processes by types of functionality on page 52. Second, it comprises a self-contained problem scope, which relates to a basic level of detail as elaborated in table 5.3 Hierarchical levels of detail for functional size measurement patterns on page 55 And third, it can be tailored according to the criteria given by Definition **DEFINITION 5.5** on page 66 to finally ensure that it represents a unique and thereby independent set of software functionality.

Problem-based functional size measurement patterns represent the missing link that integrates functional size measurement and software requirements engineering [154] based on patterns. They provide for a consistent problem description, which identifies the constituent parts of a requirements statement that are relevant for determining its functional size [104]. In addition, they provide a strategy to reuse, which mitigates quality issues related to a requirements specification [169, page 1]. Thus, these patterns for measurable problems ultimately improve the quality of the size estimates [154] for requirements.

6. Problem-Based Estimating Method

6.1. Introduction

This chapter designs a method for determining the functional size of measurable problems. It applies functional size measurement patterns as developed in the previous chapter 5 Problem-Based Units of Measure to fit desired software functionality with requirements work packages. These undergo a counting process, to obtain a function point value for a set of requirements.

Section 6.2 Background gives in a nutshell the concepts not previously presented but worked with in the following.

Section 6.3 Requirements Sizing Method gives the method for problem-based estimating, which is named Frame Counting Agenda. Its projected application is to become a substitute for the planning poker game as a means to estimate requirements by using point values. It introduces a pattern-based counting process, which makes use of the (frame diagram) structuring that underlies a requirements work package, cf. figure 5.5 on page 47. In addition, it comes with important validation conditions, that safeguard a consistent functional size measurement procedure for requirements, which is in accordance with ISO/IEC 20926:2009.

Section 6.4 Step-By-Step Guide to the Requirements Sizing Method executes the method for measuring a problem's functional complexity step-by-step. It provides by use of a running example further insights to the proposed approach.

In order to recapitulate the key principles behind the counting process and to provide a kind of evaluation for the newly created pattern-led, problem-based functional size measurement method, section 6.5 Sample Application to Jackson's Basic Frames illustrates the method in brief for some well-established problem patterns from Jackson [128].

Section 6.6 Discussion & Related Work places the newly developed method for problem-based estimating in the context of related work. It discusses the pros and cons of embedding problem patterns into a standard procedure for functional size measurement in order to stabilize requirement estimates.

Section 6.7 Summary summarizes the added-value created by this problem-based estimating method to obtain reproducible size measures for comparable requirement units.

6.2. Background

This section gives the scientific background for developing a problem-based functional size measurement method, which builds on patterns for requirements estimating as designed in chapter II Problem-Based Project Estimating.

It motivates the use of these patterns for creating a requirements model, which takes the role of a “proxy” when determining the requirement’s functional size in section 6.2.1. Then in section 6.2.2, it illustrates the necessary customizations to a functional size measurement approach, that promise for ensuring consistent function point counting based on the developed pattern-based requirements model.

Section 6.2.1 presents the core ideas behind the Proxy-Based Estimating method (PROBE) as developed by Humphrey [114, chapter 5]. It arose at the same time as Jackson’s problem frames approach [128], but it is to date not linked with patterns to form an early software estimation approach in problem-oriented software engineering.

Section 6.2.2 gives the activities and steps of the IFPUG FSM Method ISO/IEC 20926:2009 – Measurement Process [117, chapter 5], which describes a method for executing function point analysis. In addition, it outlines the adaption of this international standard to create a problem-based functional size measurement approach, which makes use of patterns as proxies for estimating the functional size of requirements.

6.2.1. Proxy-Based Estimation – The PROBE Method

Estimating of software requirements as discussed in section 5.2.2 Early Software Measurement provides for an early quantification of these, which is needed for and whose quality is of high importance to their planning in software projects. “Most size measures [...] are not available during planning[. This motivates the use of] a proxy[, which] is a stand-in measure that relates product size to planned functionality and provides a means in the planning phase for judging (and therefore, for estimating) a product’s likely size.” [177, page 33, section 3.4.1]

Following McConnell [152, Page 135ff], proxy-based estimation can make respective data available. It is characterized by:

1. Identifying and counting a proxy that correlates with *what you really want to estimate*[1].
2. Estimate the quantity *of what you are ultimately interested in* on the basis of historical data available to you from counted proxies.

As presented by table 6.1, a good proxy is the key. It not only enables the gathering of data, but it also improves the quality of these data and therefore ensures a more consistent estimating and planning processes.

Problem-based functional size measurement utilizes requirements work packages as proxy, which are built on patterns, and whose size measurement is guided by validation conditions to guarantee consistent counts. These patterns create requirement proxies, which procure a “form [...] for gathering and retaining [size] data” [177, page 14, section 1.2.3].

Within the Personal Software Process (PSP) framework [177, page 34, Knowledge Area 3.5], Humphrey developed a Proxy-Based Estimating method (PROBE) [114, chapter 5], which enables software engineers in a disciplined way [115] to estimate size and effort of their work based on historical data from the proxies.

PSP is a step-by-step self-improvement approach, which enables the gathering of performance data needed for understanding and enhancing the individual’s productivity. PROBE makes use of

[1]Note: In this work this quantity relates to functional *requirements*.

these “personal data to judge a new program’s size and required development time” [115, page 81], and to feature statistical estimating and planning methods.

- ☑ Proxy *size measurement* should closely relate to product *development effort*
- ☑ Proxy content of a product should be *automatically countable*
- ☑ Proxy should be *easy to visualize at the beginning* of the project
- ☑ Proxy should be *customizable* to the special needs of its using organization
- ☑ Proxy should be *sensitive to any implementation variations* that impact development effort
- ☐ Proxy prediction quality should be determined by use of correlation method

Legend: ☑= the patterns provided here meet this criterion for a good proxy

TABLE 6.1 Criteria for selecting a good proxy, adapted from [114, p. 111] and [177, p. 33]

The use of a proxy in PROBE accounts for the idea, that developing software functionality, which is (of) similar (size) compared to one developed in the past, will require the same effort. It applies program objects as proxies that reflect real-world entities, but their size measurement is executed on development artifacts, i.e. program parts, which become available late in a project.

Size data in PROBE relates to “developed product size in LOC” [115, page 81]. Effort refers in PROBE to the time needed for delivering the respective LOC, i.e. for completing the development of a defined software part.

“Other proxies [...] are possible” [115, page 81], of which “The function-point method is obviously a candidate” to Humphrey [114, page 113]. This is approached in here for providing a means to early software size measurement, which is designed to operate even in the absense of historical data.

6.2.2. IFPUG FSM Method ISO/IEC 20926:2009 – Measurement Process

The International Function Point Users Group (IFPUG) standard [117, chapter 5] describes a functional size measurement method, which is not limited to but applicable in the early phases of software projects. It comprises a stepwise process with related counting rules, which allow for measuring the "determinants of size" as given in a requirements specification and introduced in section 5.2.3 IFPUG FSM Method ISO/IEC 20926:2009 – Terms and Definitions. The measurement result is represented by a numerical value given in so-called function points (FP), which represents the amount of functionality involved with the requirements under consideration.

Figure 6.1 illustrates the process of functional size measurement as given by IFPUG standard [117, chapter 5], and its customization by the pattern-based Requirements Sizing Method as introduced in section 6.3 Requirements Sizing Method.

On its left side, it shows the ten activities of the functional size measurement process as given in chapter 5 of the IFPUG standard IFPUG [117]. The flow diagram is taken from IFPUG [117, figure 1] for illustrating the initial IFPUG measurement method, and what has been economized for its adaption to a problem-based functional size measurement approach.

In the middle, figure 6.1 gives more detail on the activities and thus the structuring of the IFPUG functional size measurement process. Every modification to this measurement method is emphasized through a coloring in gray.

On its right side, figure 6.1 shows the coverage of activities for the IFPUG method and the Frame Counting Agenda (FCA). This problem-based functional size measurement method takes a requirements work package (RWP) as input, and assists the requirements estimator in determining the functional size of its involved functional user requirements (FUR). The output of the FCA is a numerical value, which answers the functional size question by a number giving the function points for the FUR of the respective requirements work package, and thus is attached to it.

As can be seen, the entire measurement process is supported by problem-based functional size measurement patterns as designed in section 5.6 of this work and summarized in table 5.5. That is, these patterns for classifying requirements are independent of the requirements sizing method, but functional size measurement according to the FCA depends on their application. This is due to the fact, that each measurement step within the FCA is accompanied by validation conditions as introduced in table 6.3, which safeguard the consistency of the counting process and operate on the pattern that underlies a requirements work package. Each validation condition represents an interpretation of the function point counting rules as given in the IFPUG standard ISO 20926, such that these rules become applicable in problem-based functional size measurement.

In detail...

The activities *5.1*, *5.4.1*, and *5.5.1* of the IFPUG FSM method present only an overview of their respective subactivities, which is why these are marked in gray and mentioned for the sake of completeness only in figure 6.1.

Activity *5.10.2* considers the fact, that the FCA is a customization of the IFPUG method. In consequence activity *5.10.1*, which cares about application of the original IFPUG method, is not of relevance in here and thus marked in gray, too.

Problem-based functional size measurement patterns form the basis for requirements work packages, cf. figure 5.5 Conceptualization of a Requirements Work Package. Their use is not limited to requirements estimating, they are also advantageous to the project planning. So, creating measurable problems and packing these into a requirements work package is on the one side a prerequisite for utilizing the FCA, but on the other side this activity of grouping requirements into measurable units is not necessarily part of a requirements estimating method. This means, activity *5.2 Gathering available documentation* has become an obsolete one for the FCA, which is why it is grayed-out. Requirements work packages as applicable in the proposed problem-based requirements sizing approach by means of patterns include all information that is of relevance for executing their functional size measurement. Accordingly, a requirements work package is an input to FCA.

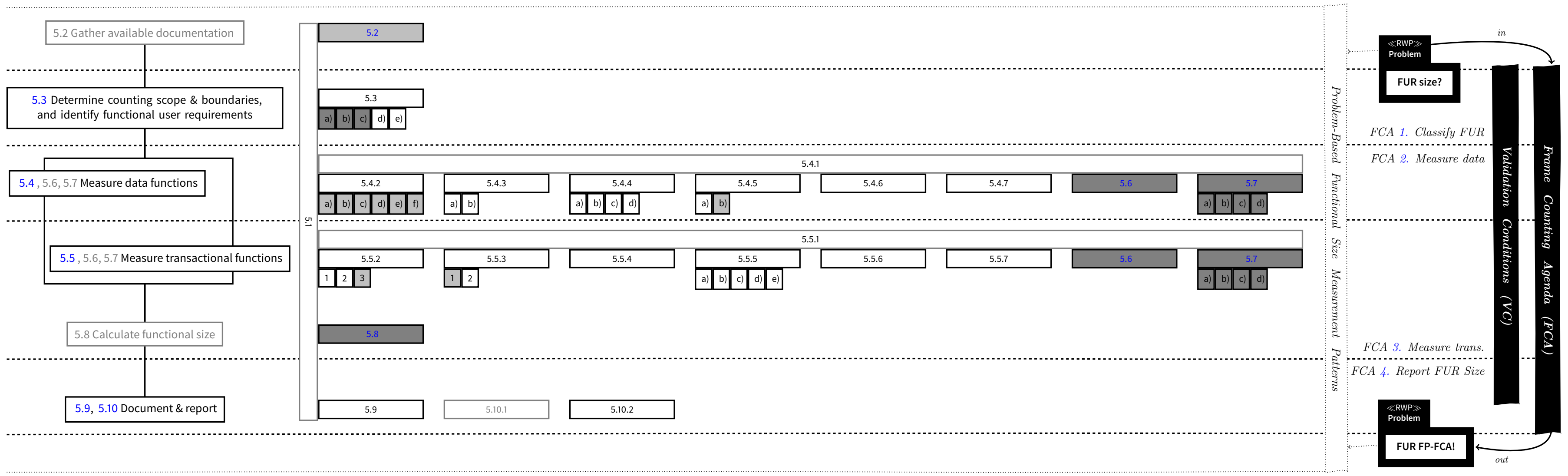
Activities *5.6* and *5.7* are repeated ones within *5.4 Measure data functions* and *5.5 Measure transactional functions*, where these activities serve as preparation for activity *5.8 Calculate functional size*. These three activities *5.6*, *5.7*, and *5.8* of IFPUG FSM have become obsolete within the frame counting agenda and consequently are marked in gray. The reason behind this circumstance is, that IFPUG FSM activity *5.8* provides several formulas to calculate the functional size for requirements, based on the purpose of the count, its type and scope, which have been determined in activity *5.3.a)*, *5.3.b)*, and *5.3.c)*. These different formulas address the fact, that the functional size of requirements is intented for different uses. For instance, in development projects it indicates added software functionality, in enhancement projects it indicates only modified software functionality. This is of no relevance to the frame counting agenda. It is applied for determining the functional size implemented by one requirements work package. How the resulting function points are used is not in the responsibility of the requirements sizing method. Consequently, activities *5.8* of IFPUG FSM method, and its related activities *5.6* and *5.7*, as well as *5.3.2a)* to *5.3.2c)* become obsolete[2] in the FCA, and are marked by a gray-shaded box in figure 6.1.

It is the use of problem-based functional size measurement patterns for structuring requirements work packages, which entail that IFPUG FSM activities *5.2* and *5.6*, *5.7*, and *5.8* can be skipped in the FCA, and why activity *5.4.2*, *5.4.5a)*, *5.5.2*, and *5.5.3* benefits from their presence as discussed in appendix B.1.

In summary, the frame counting agenda focuses on IFPUG FSM activities 5.3, 5.4, 5.5, 5.9, and 5.10 for implementing a problem-based requirements sizing method. It provides an approach to estimate requirements reproducibly, and it allows for determining consistent function points for a requirements work package, which are in accordance with the IFPUG counting regime.

Details on the interpretation of the IFPUG standard for problem-based functional size measurement are given in the appendix B.1. It presents a brief justification of how the Validation Conditions that safeguard the function point counting have been deduced for becoming applicable to software requirement patterns.

[2]Note: This leaving out of activities should not be confused with an ignorance of these. Problem-based functional size measurement patterns allow for an optimization of the IFPUG FSM process, taking lean principles as listed in appendix C into account. These principles value the elimination of waste, e.g. by removing unproductive repetitions, or by improving the quality of a process through standardization, which in this case is enabled by means of patterns for software requirements. These functional size measurement patterns make the repetitive processing of activities *5.4* and *5.5* much more controllable and straightforward, since their use prevents the arbitrary picking of any "determinant of size" in the available requirements documentation.


5.2 Gather available documentation
5.3 Determine counting scope & boundaries, and identify functional user requirements
5.4, 5.6, 5.7 Measure data functions
5.5, 5.6, 5.7 Measure transactional functions
5.8 Calculate functional size
5.9, 5.10 Document & report
5.1
5.2
5.3
a) b) c) d) e)
5.4.1
5.4.2
5.4.3
5.4.4
5.4.5
5.4.6
5.4.7
5.6
5.7
a) b) c) d) e) f)
5.5.1
5.5.2
5.5.3
5.5.4
5.5.5
5.5.6
5.5.7
1 2 3
5.8
5.9
5.10.1
5.10.2
Problem-Based Functional Size Measurement Patterns
«RWP» Problem
FUR size?
in
FCA 1. Classify FUR
FCA 2. Measure data
FCA 3. Measure trans.
FCA 4. Report FUR Size
Validation Conditions (VC)
Frame Counting Agenda (FCA)
«RWP» Problem
FUR FP-FCA!
out

Legend:

▩ N.N gray-shaded box marks an activity, whose use is obsolete within the frame counting agenda

☐ N.N gray-colored box or text marks an activity, which is mentioned only for the sake of completeness

FIGURE 6.1 Towards a more lean function point counting process, cf. [117, p. 9, fig. 1]

6.3. Requirements Sizing Method

Problem-based units of measure as introduced in the previous chapter 5 Problem-Based Units of Measure are a means to establish a uniform problem description for a measurable set of requirements. This chapter gives a method that allows for determining the functional size of a requirements unit in a reproducible way.

In the focus of considerations is a customized function point counting process as described in section 6.3.1 Counting Process that is enriched with validation conditions as detailed in section 6.3.2 Validation Conditions. These safeguard the proper interpretation of the Base Functional Components (BFCs) involved with a measurable problem, and thereby care for consistent application of associated counting rules in accordance with ISO/IEC 20926:2009.

Since the requirements sizing method operates on the constituent parts of a functional size measurement pattern, i.e. the structuring given by one of the problem frames in table 5.5 used for instantiating a requirements work package, its process description as given in table 6.2 is called Frame Counting Agenda.

6.3.1. Counting Process

Table 6.2 gives the frame counting procedure developed in this work. It takes the agenda concept [106] to adopt each step of the measurement process [117, section 5, pages 8–19, and 21] defined by ISO/IEC 20926:2009 to a pattern-based functional size measurement method for requirements as follows.

The process description of the counting procedure is structured according to the agenda concept into input, activities, output, and validation conditions. The **input** lists all the information, which undergo a change by the enumerated **activities**. The **validation conditions** place constraints on the activities, such that the change of information results in an **output**, which features these conditions. That is, the quality of the counting procedure does not solely rely on a correct execution of the measurement activities, it also depends on the observance of the validation conditions, which define the rules of the counting game.

The requirements get classified by functional size measurement patterns, each of which supports the identification of all the base functional components involved in this set of software functionality, and which can be measured, respectively. Step-by-Step and by means of validation conditions and ISO/IEC 20926:2009 complexity and size tables, each constituent part of this measurable problem is assigned a point value, which cumulates to a functional size measure for the entire requirements work package.

Name:	Counting Procedure
Input:	· FUR::={requirement statement(s)} · Functional size measurement patterns, such as problem frames in table 5.5 on page 72 · Data function complexity and size tables, in ISO/IEC 20926:2009 [117, A.1 and A.2, page 23] · Transactional function complexity and size tables, in ISO/IEC 20926:2009 [117, A.3–A.5, page 23]
Activities:	*1. Classify FUR by Functional Size Measurement Patterns.* *2. Determine Data Functions.* 2.a Identify problem domains as data functions. 2.b Classify data functions into ILF or EIF. 2.c Count DET for each data function. 2.d Count RET for each data function. 2.e Determine functional complexity for data functions. 2.f Determine functional size for data functions. *3. Determine Transactional Function.* 3.a Identify machine domain as transactional function. 3.b Classify transactional function as either EI, EQ, or EO. 3.c Count FTR for transactional function. 3.d Count DET for transactional function. 3.e Determine functional complexity for transactional function. 3.f Determine functional size for transactional function. *4. Report Functional Size for FUR.*
Output:	· Requirements work package and its size in function points
Validation:	· *continued in section 6.3.2 on page 83*

TABLE 6.2 Frame Counting Agenda

6.3.2. Validation Conditions

Table 6.3 enumerates all validation conditions needed within the frame counting agenda to establish a consistent measurement process.

It takes the rules for functional size measurement included in ISO 20926 [117] and adapts these to work with functional size measurement patterns as developed in chapter 5. Hence, these validation conditions are subject to interpretation of the ISO standard and the conception of problem frames. The preceding chapters and sections deduce and discuss the interpretation used in this work in detail. Table 6.3 gives the result of this interpretation, which is executable on problem frames, cf. figure 5.5 on page 47. The appendix B Sanity Checks gives the reasoning behind this interpretation of the ISO 20926 measurement process.

The validation conditions in table 6.3 are sorted with respect to their use within the different activities of the requirements sizing method. Nevertheless, they hold for the entire measurement process, i.e. their effect is beyond the activity they are used in.

Validation:		
Activity 1.	*Classify FUR by Functional Size Measurement Patterns.*	
	V.i	These validation conditions apply to the consideration of one measurable problem, i.e. problem diagram.
	V.ii	A problem frame with only one constrained problem domain and at least one referenced problem domain has been applied to set up the problem diagram.
	V.iii	Only shared phenomena at the machine interface of the problem diagram are considered.
Activity 2.a	*Identify problem domains as data functions.*	
	V.iv	A problem domain with symbolic phenomena can take the role of a data function, i.e. an ILF or an EIF.
	V.v	A problem domain with no symbolic phenomena cannot take the role of a data function, i.e. an ILF or an EIF.
Activity 2.b	*Classify data functions into ILF or EIF.*	
	V.vi	A constrained domain can only take the role of an ILF.
	V.vii	A referenced domain can either take the role of an ILF or an EIF.
	V.viii	A causal domain can either take the role of an ILF or an EIF.
	V.ix	A biddable domain can only take the role of an EIF.
	V.x	A lexical or a display domain can only take the role of an ILF.

Validation:

Activity 2.c *Count DET for each data function.*

V.xi The number of DET counted for a DF corresponds to the number of its symbolic phenomena.

Activity 2.d *Count RET for each data function.*

V.xii The number of RET counted for a DF is one (1).

Activity 2.e *Determine functional complexity for data functions.*

V.xiii The DET for all n data functions classified as ILF in this measurable problem is cumulated to $ILF_{DET} = \sum_{i=1}^{n} DET_{ILF_i}$.

V.xiv The DET for all m data functions classified as EIF in this measurable problem is cumulated to $EIF_{DET} = \sum_{i=1}^{m} DET_{EIF_i}$.

V.xv If two problem domains, one is an ILF and the other an EIF, share the same k symbolic phenomena, then the respective k DET are only counted for the ILF. The EIF_{DET} is decremented by the respective number of DET, i.e. $EIF_{DET} - k$.

V.xvi If two problem domains, both ILF, share the same l symbolic phenomena, then the respective l DET are only counted for the ILF that corresponds to a constrained problem domain. The ILF_{DET} is decremented by the respective number of DET, i.e. $ILF_{DET} - l$.

V.xvii The RET for all n data functions classified as ILF in this measurable problem is cumulated to $ILF_{RET} = \sum_{i=1}^{n} RET_{ILF_i}$.

V.xviii The RET for all m data functions classified as EIF in this measurable problem is cumulated to $EIF_{RET} = \sum_{i=1}^{m} RET_{EIF_i}$.

V.xix If ILF_{RET}, ILF_{DET}, EIF_{RET}, or EIF_{DET} is zero (0), associating a DF functional complexity level is not applicable. In this case, the respective ILF or EIF complexity level becomes $ILF|EIF_{Complexity}$::={n/a}.

V.xx The DF functional complexity for all ILF in this measurable problem is determined by $ILF_{Complexity}$(ILF_{RET},ILF_{DET})::={low|average|high} according to Table A.1 in ISO 20926.

V.xxi The DF functional complexity for all EIF in this measurable problem is determined by $EIF_{Complexity}$(EIF_{RET},EIF_{DET})::={low|average|high} according to Table A.1 in ISO 20926.

Validation:

Activity 2.f	*Determine functional size for data functions.*	
	V.xxii	If no DF functional complexity $ILF/EIF_{Complexity}$={n/a} is applied, the respective DF functional size ILF/EIF_{Size} is considered as zero (0) function points.
	V.xxiii	The DF functional size for all ILF in this measurable problem is given in function points and determined by $ILF_{Size}(ILF_{Complexity}, ILF)$ according to Table A.2 in ISO 20926.
	V.xxiv	The DF functional size for all EIF in this measurable problem is given in function points and determined by $EIF_{Size}(EIF_{Complexity}, EIF)$ according to Table A.2 in ISO 20926.
Activity 3.b	*Classify transactional function as either EI, EQ, or EO.*	
	V.xxv	The classification of the transactional function TF$_{Type}$::={EI|EQ|EO} is justified by the applied problem frame, and aligned with *Activity 1.*, validation condition V.ii.
Activity 3.c	*Count FTR for transactional function.*	
	V.xxvi	The number of FTR counted for a TF corresponds to the number of DFs in this measurable problem: TF_{FTR} = n ILF + m EIF.
Activity 3.d	*Count DET for transactional function.*	
	V.xxvii	The number of DET counted for a TF corresponds to its shared, symbolic as well as causal phenomena.
	V.xxviii	Shared phenomena with a lexical domain do not cross the application boundary. They do not count in TF_{DET}.
	V.xxix	Shared phenomena with a causal, biddable or display domain do cross the application boundary. They count in TF_{DET}.
	V.xxx	Each causal phenomenon that crosses the application boundary adds one (1) DET for the data element types of a transactional function TF_{DET}.
	V.xxxi	Each symbolic phenomenon that crosses the application boundary adds one (1) DET for the data element types of a transactional function TF_{DET}.

Validation:		
Activity 3.e	*Determine functional complexity for transactional function.*	
	V.xxxii	The TF functional complexity in this measurable problem is determined by $TF_{Complexity}(TF_{Type}, TF_{FTR}, TF_{DET})$::={low|average|high} according to the Table A.3 for EI, or Table A.4 for EO and EQ in ISO 20926.
Activity 3.f	*Determine functional size for transactional function.*	
	V.xxxiii	The TF functional size in this measurable problem is given in function points and determined by $TF_{Size}(TF_{Complexity}, TF_{Type})$ according to Table A.5 in ISO 20926.
Activity 4.	*Report Functional Size for FUR.*	
	V.xxxiv	The functional size for a measurable problem is reported in function points (FP), following the format: <Measurable Problem$_{size}$> FP (IFPUG-ISO/IEC 20926:2009-FCA), which indicates by the postfix of –FCA (for Frame Counting Agenda) a customization of the IFPUG standard. The functional size of a measurable problem is a cumulated value of DF and TF sizes given in function points: $Measurable\ Problem_{size} = ILF_{Size} + EIF_{Size} + TF_{Size}$.

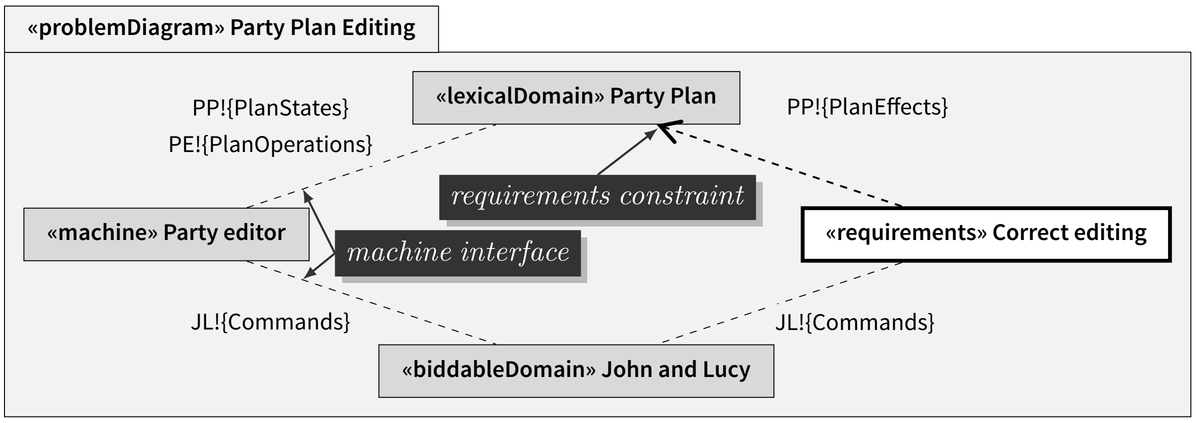
TABLE 6.3 Frame Counting Agenda, Validation Conditions

6.4. Step-By-Step Guide to the Requirements Sizing Method

This section illustrates how to execute the frame counting agenda as given in table 6.2 on page 82 in detail. It makes use of Jackson's Party Plan Editing problem [128, page 98] as a running example. Supplemental comments, which are given in black boxes, and validation conditions relevant to each activity of the counting process are added by way of explanation.

EXAMPLE 6.1 Frame Counting Agenda – Introduction to Jackson's Party Plan Editing problem

This problem diagram gives Jackson's Party Plan Editing problem in a UML stereotype notation as applied in the UML4PF tool [107]. As can be seen by the requirements constraint and the involved domain types, this problem fits to the "simple workpieces" problem frame, which belongs to the set of functional size measurement patterns as defined in table 5.5 on page 72.



The shared phenomena at the machine interface require more detail than provided by Jackson's original problem diagram, such that the requirements sizing method can become effective. That is, causal and symbolic phenomena should be distinguishable, such as *PlanStates* and *PlanOperations*, and grouped phenomena should be indicated in such a way that they become countable, such as *Commands*. Therefore, the requirement *Correct editing* is refined, before the counting procedure starts, and the problem diagram is updated, respectively.

The following requirement **CE_FUR 01** as instance of *Correct editing* is created based on the discussion in Jackson [128, pages 125–129].

CE_FUR 01 John and Lucy plan a party to which they want to invite some guests.

This FUR allows for deriving a specification at the machine interface of the domain *Party editor* that is as follows:

CE_Spec 01.1 If John and Lucy plan a party, a party plan is created, which has a specific party motto.

CE_Spec 01.2 If John and Lucy plan to invite some guests to a party, theses guests are registered to the specific party plan.

CE_FUR 01, CE_Spec 01.1, and CE_Spec 01.2 are used to detail the shared phenomena for user *commands* as well as *plan operations*, *states*, and *effects* in the following problem description for Jackson's Party Plan Editing.

Activity 1. Classify FUR by Functional Size Measurement Patterns

This activity serves to set up requirements work packages as defined on page 46 by making use of the functional size measurement patterns as given in table 5.5 on page 72. These patterns, i.e. a set of selected problem frames, allow for a grouping of requirements to a measurable problem given by a respective problem diagram, which is also appropriate for measuring the requirements' functional size.

The following validation conditions safeguard this activity:

V.i These validation conditions apply to the consideration of one measurable problem, i.e. problem diagram.

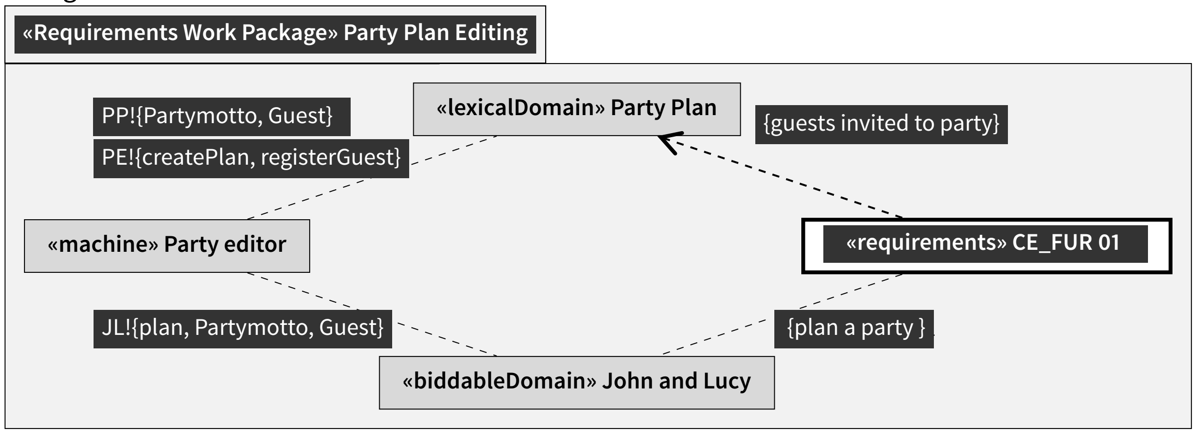
V.ii A problem frame with only one constrained problem domain and at least one referenced problem domain has been applied to set up the problem diagram.

V.iii Only shared phenomena at the machine interface of the problem diagram are considered.

Validation conditions **V.i** and **V.ii** ensure the use of Basic Problem Frames for this counting procedure. This is fulfilled, if only functional size measurement patterns as defined in table 5.5 are applied. Validation condition **V.iii** serves as a reminder for the requirements measurer to care for the shared phenomena at the machine interface in order to execute the counting procedure.

EXAMPLE 6.2 Frame Counting Agenda – Activity 1. Classify requirements by patterns

This problem diagram shows Jackson's Party Plan Editing problem, which incorporates **CE_FUR 01**, **CE_Spec 01.1**, and **CE_Spec 01.2** by means of a "simple workpieces" frame to a requirements work package.


«Requirements Work Package» Party Plan Editing
PP!{Partymotto, Guest}
PE!{createPlan, registerGuest}
«lexicalDomain» Party Plan
{guests invited to party}
«machine» Party editor
«requirements» CE_FUR 01
JL!{plan, Partymotto, Guest}
{plan a party }
«biddableDomain» John and Lucy

The black boxes indicate the alterations done for advancing the diagram for the requirements set of *Correct editing* by CE_FUR 01, CE_Spec 01.1, and CE_Spec 01.2 to a measurable problem, i.e. a requirements work package, which allows for distinguishing the types of domains and shared phenomena that are in the counting scope. The validation conditions **V.i** to **V.iii** for this counting process activity 1. hold.

Activity 2. Determine Data Functions

This activity serves to determine the amount of data involved in the requirements work package. The input to this activity requires a framed set of requirements, such as given by the requirements work package for “Party Plan Editing” and the data function complexity and sizes tables as defined in ISO 20926 and arranged on pages 262ff in the appendix of this work. Then, six steps 2.a to 2.f have to be executed in order to identify and count relevant data information. At the end of this activity 2., the functional size for the data functions within this requirements work package is calculated and given in function points.

2.a Identify problem domains as data functions

This step serves to locate the data that must be counted. It must be decided, which problem domains are considered as data functions within this requirements work package.

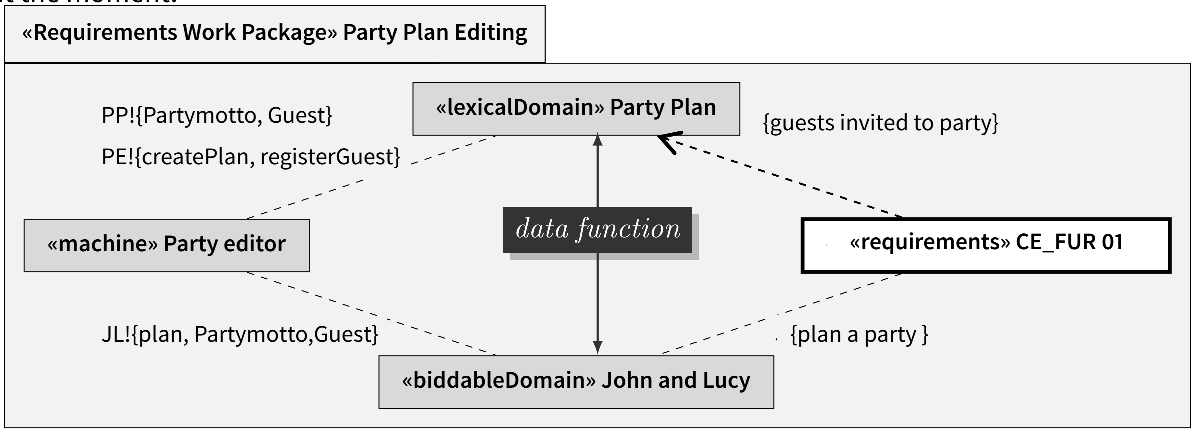
The following validation conditions safeguard this activity:

V.iv A problem domain with symbolic phenomena can take the role of a data function, i.e. an ILF or an EIF.

V.v A problem domain with no symbolic phenomena cannot take the role of a data function, i.e. an ILF or an EIF.

EXAMPLE 6.3 Frame Counting Agenda – Activity 2./Step 2.a Determine data functions

Both problem domains in this requirements work package are associated with some symbolic phenomena at their machine interface. *Party Plan* and *John and Lucy* are equally concerned with *Partymotto* and *Guest*. These shared phenomena represent data information to be gathered and processed by the *Party editor* machine.
According to table 5.1 on page 48, the shared phenomena *plan* and *createPlan* are causal phenomena, which represent some control mechanism on the symbolic phenomena. They constitute commands for triggering data transformation and manipulation. Causal phenomena are not of relevance at the moment.


«Requirements Work Package» Party Plan Editing
PP!{Partymotto, Guest}
PE!{createPlan, registerGuest}
«lexicalDomain» Party Plan
{guests invited to party}
«machine» Party editor
data function
«requirements» CE_FUR 01
JL!{plan, Partymotto,Guest}
{plan a party }
«biddableDomain» John and Lucy

That is why this requirements work package has two data functions, namely the problem domains *Party Plan* and *John and Lucy*. The validation conditions V.iv to V.v for this counting process activity hold.

2.b Classify data functions into ILF or EIF

This steps serves to classify the data involved with this requirements work package with respect to the application boundary. It must be decided, if a data function resides within the application boundary, such as an internal logical file (ILF), or if it resides outside the application boundary, such as an external interface file (EIF), cf. figure 5.3 on page 43.

The following validation conditions safeguard this activity:

V.vi A constrained domain can only take the role of an ILF.

V.vii A referenced domain can either take the role of an ILF or an EIF.

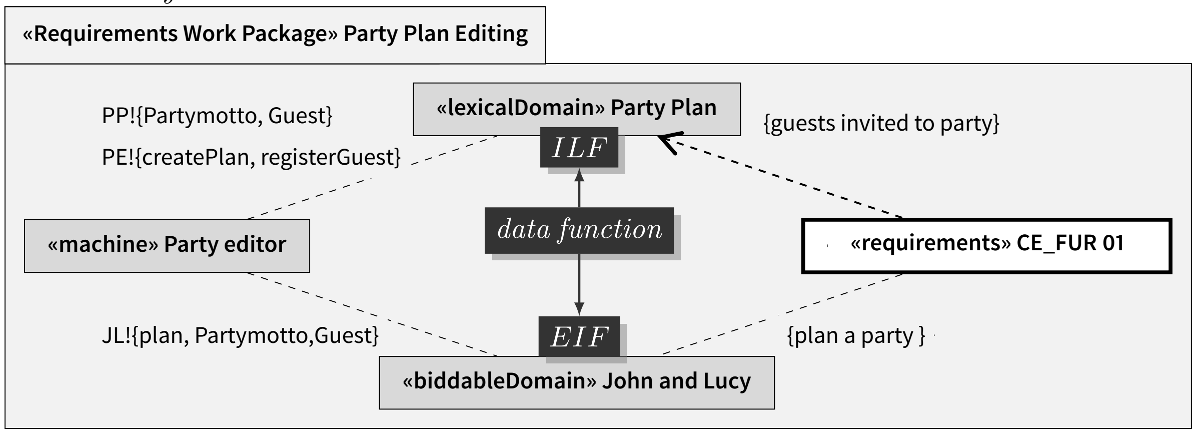
V.viii A causal domain can either take the role of an ILF or an EIF.

V.ix A biddable domain can only take the role of an EIF.

V.x A lexical or a display domain can only take the role of an ILF.

EXAMPLE 6.4 Frame Counting Agenda – Step 2.b Classify data functions

There are two data functions in this requirements work package, namely *Party Plan* and *John and Lucy*.


«Requirements Work Package» Party Plan Editing
PP!{Partymotto, Guest}
PE!{createPlan, registerGuest}
«lexicalDomain» Party Plan
ILF
{guests invited to party}
«machine» Party editor
data function
«requirements» CE_FUR 01
JL!{plan, Partymotto,Guest}
EIF
{plan a party }
«biddableDomain» John and Lucy

Party Plan is a constrained, lexical problem domain. According to validation conditions V.vi and V.x it can only take the role of an internal logical file (ILF). *John and Lucy* is a referenced, biddable problem domain. According to validation conditions V.vii and V.ix it can only take the role of an external interface file (EIF). The validation conditions for this counting process activity hold.

2.c Count DET for each data function

This step serves to identify the data information or data element types (DET) respectively, which are involved with a data function. In case of recurring information in one requirements work package, it must be decided, whether or not to count these, in order to avoid miscounts. Respective validation conditions are provided in step **2.e** to prevent redundant DET counts.

The following validation condition safeguards this activity:

V.xi The number of DET counted for a DF corresponds to the number of its symbolic phenomena.

The respective example is included to the following counting process step 2.d.

2.d Count RET for each data function

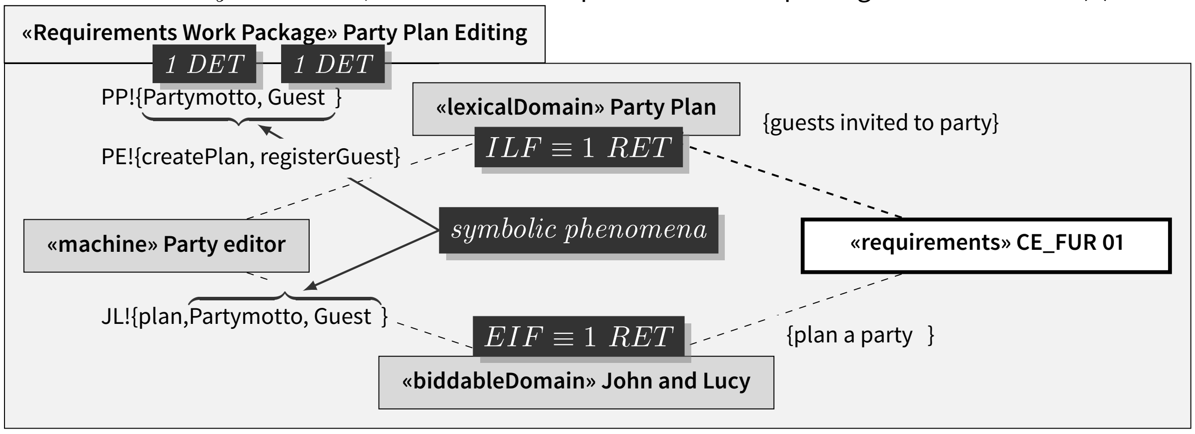
This step serves to identify the number of data sources or respective record element types (RET) within a requirements work package. It must be decided, how many interfaces need to be considered by the machine in order to establish desired control, i.e. software functionality.

The following validation condition safeguards this activity:

V.xii The number of RET counted for a DF is one (1).

EXAMPLE 6.5 Frame Counting Agenda – Step 2.c and 2.d Count data and record element types

There are two data functions in this requirements work package, namely $Party\ Plan$ and $John\ and\ Lucy$. With respect to step 2.c, both hold the same symbolic phenomena at the machine interface, namely $Partymotto$ (1 DET) and $Guest$ (1 DET). Accordingly, the data function $Party\ Plan$ has 2 DET, and the data function $John\ and\ Lucy$ has also 2 DET at their machine interface.
Since only problem domains that are involved with some symbolic phenomena at the machine interface can take the role of a data function (see V.iv), each of these is associated with a RET. $Party\ Plan$ and $John\ and\ Lucy$ are two DF, such that this requirements work package has in sum two (2) RET.


«Requirements Work Package» Party Plan Editing
1 DET
1 DET
PP!{Partymotto, Guest }
PE!{createPlan, registerGuest}
«lexicalDomain» Party Plan
ILF ≡ 1 RET
{guests invited to party}
«machine» Party editor
symbolic phenomena
«requirements» CE_FUR 01
JL!{plan,Partymotto, Guest }
EIF ≡ 1 RET
{plan a party }
«biddableDomain» John and Lucy

The validation conditions V.xi and V.xii for the counting process activities 2.c and 2.d hold.

2.e Determine functional complexity for data functions

This step serves to determine the functional complexity of a requirements work package related to its data functions. Their number of DET and the RET is used as parameter to the ISO 20926 data function complexity tables and results in a measure of functional complexity, which is either low, average, or high.

The following validation conditions safeguard this activity:

V.xiii The DET for all n data functions classified as ILF in this measurable problem is cumulated to $ILF_{DET} = \sum_{i=1}^{n} DET_{ILF_i}$.

V.xiv The DET for all m data functions classified as EIF in this measurable problem is cumulated to $EIF_{DET} = \sum_{i=1}^{m} DET_{EIF_i}$.

V.xv If two problem domains, one is an ILF and the other an EIF, share the same set of k symbolic phenomena, then the respective k DET are only counted for the ILF. The EIF_{DET} is decremented by the respective number of DET, i.e. $EIF_{DET} - k$.

V.xvi If two problem domains, both ILF, share the same set of l symbolic phenomena, then the respective l DET are only counted for the ILF that corresponds to a constrained problem domain. The ILF_{DET} is decremented by the respective number of DET, i.e. $ILF_{DET} - l$.

V.xvii The RET for all n data functions classified as ILF in this measurable problem is cumulated to $ILF_{RET} = \sum_{i=1}^{n} RET_{ILF_i}$.

V.xviii The RET for all m data functions classified as EIF in this measurable problem is cumulated to $EIF_{RET} = \sum_{i=1}^{m} RET_{EIF_i}$.

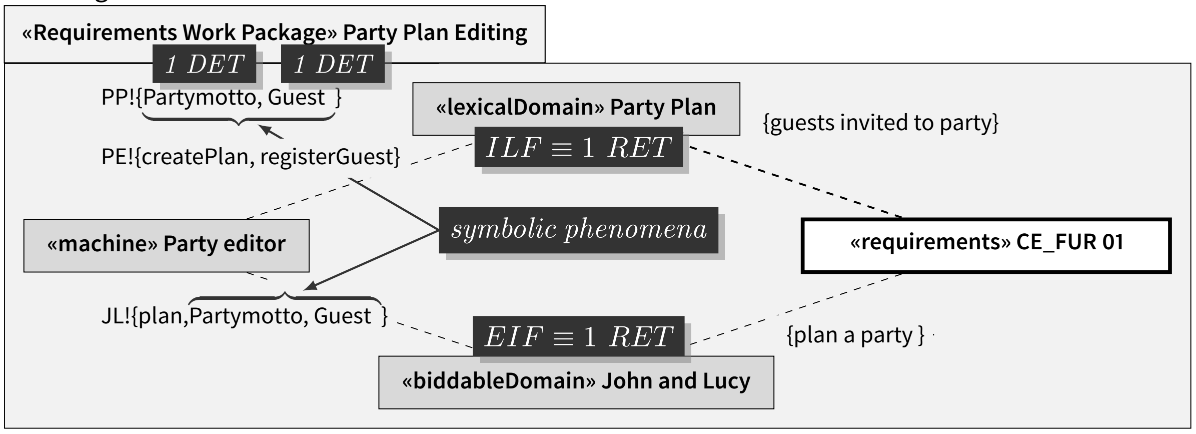
V.xix If ILF_{RET}, ILF_{DET}, EIF_{RET}, or EIF_{DET} is zero (0), associating a DF functional complexity level is not applicable. In this case, the respective ILF or EIF complexity level becomes $ILF|EIF_{Complexity}$::={n/a}.

V.xx The DF functional complexity for all ILF in this measurable problem is determined by $ILF_{Complexity}$(ILF_{RET},ILF_{DET})::={low|average|high} according to Table A.1 in ISO 20926.

V.xxi The DF functional complexity for all EIF in this measurable problem is determined by $EIF_{Complexity}$(EIF_{RET},EIF_{DET})::={low|average|high} according to Table A.1 in ISO 20926.

EXAMPLE 6.6 Frame Counting Agenda – Step 2.e Determine data function complexity

$Party\ Plan$ is the only data function in this requirements work package, which takes the role of an ILF. According to **V.xiii**, the symbolic phenomena $Partymotto$ and $Guest$ at the machine interface of $Party\ Plan$: $DET_{ILF} = 1 + 1 = ILF_{DET} = 2$.
Since $Party\ Plan$ is the only ILF in this requirements work package, $RET_{ILF} = ILF_{RET} = 1$ according to **V.xvii**.
$John\ and\ Lucy$ is the only data function in this requirements work package, which takes the role of an EIF. According to **V.xiv**, its symbolic phenomena $Partymotto$ and $Guest$ at the machine interface count as follows $John\ and\ Lucy$: $DET_{EIF} = 1 + 1 = EIF_{DET} = 2$.
Since $John\ and\ Lucy$ is the only EIF in this requirements work package, $RET_{EIF} = EIF_{RET} = 1$ according to **V.xviii**.


«Requirements Work Package» Party Plan Editing
1 DET
1 DET
PP!{Partymotto, Guest }
PE!{createPlan, registerGuest}
«lexicalDomain» Party Plan
ILF ≡ 1 RET
{guests invited to party}
«machine» Party editor
symbolic phenomena
«requirements» CE_FUR 01
JL!{plan,Partymotto, Guest }
EIF ≡ 1 RET
{plan a party }
«biddableDomain» John and Lucy

Since the problem domains $Party\ Plan$ and $John\ and\ Lucy$ are concerned with the same set of symbolic phenomena, validation condition **V.xv** comes into play. Having in mind that activity 2. is about determining internal data storage requirements, the recurring $k = 2 = DET_{EIF}$ phenomena are only considered for the ILF $Party\ Plan$, but not for the EIF $John\ and\ Lucy$. According to **V.xv**, the DET value counted for the EIF changes to $EIF_{DET} = 0$ after this step 2.e, $ILF_{DET} = 2$ remains.
Due to validation conditions **V.xix**, and **V.xxi**, the functional complexity for the EIF $John\ and\ Lucy$ cannot be determined, it becomes $EIF_{Complexity}(EIF_{RET}, EIF_{DET}) = (1, 0) = \{n/a\}$.
Due to validation condition **V.xx** the functional complexity for the ILF $Party\ Plan$ becomes $ILF_{Complexity}(ILF_{RET}, ILF_{DET}) = (1, 1) = \{low\}$ according to the data function complexity matrix in ISO 20926 [117, Table A.1, page 23] as on page 262 in the appendix. The validation conditions for this counting process activity are meet.

2.f Determine functional size for data functions

This step serves to determine the functional size of a requirements work package related to its data functions. Its functional complexity for each data function as determined in step 2.e is used as parameter to the ISO 20926 data function size tables and results in a measure given in function points between 5 and 15.

The following validation conditions safeguard this activity:

V.xxii If no DF functional complexity $ILF/EIF_{Complexity}$={n/a} is applied, the respective DF functional size ILF/EIF_{Size} is considered as zero (0) function points.

V.xxiii The DF functional size for all ILF in this measurable problem is given in function points and determined by $ILF_{Size}(ILF_{Complexity}, ILF)$ according to Table A.2 in ISO 20926.

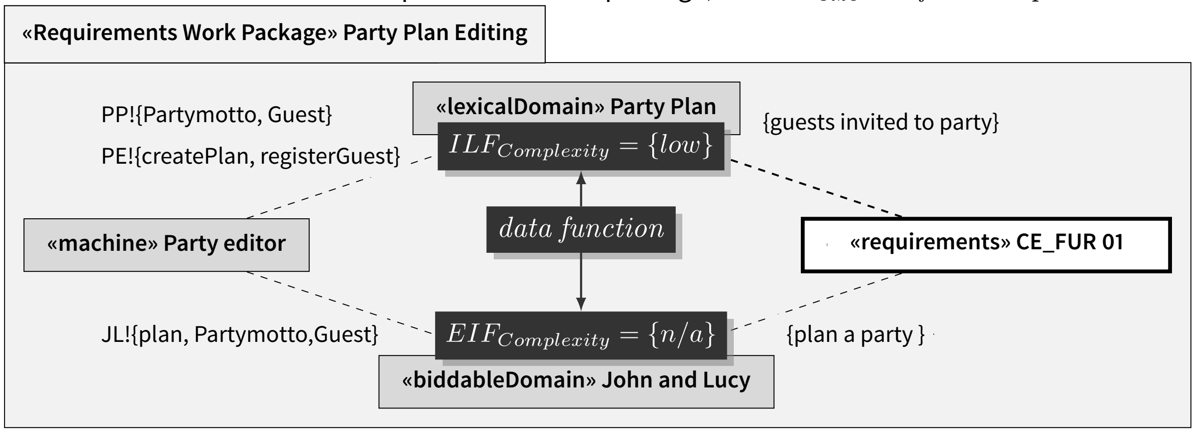
V.xxiv The DF functional size for all EIF in this measurable problem is given in function points and determined by $EIF_{Size}(EIF_{Complexity}, EIF)$ according to Table A.2 in ISO 20926.

EXAMPLE 6.7 Frame Counting Agenda – Step 2.f Determine data function size

The requirements work package "Party Plan Editing" comprises two data functions, namely $Party\ Plan$, which is an ILF having a low functional complexity, and $John\ and\ Lucy$, which is an EIF with a not determinable functional complexity.

According to validation condition **V.xxiii** and the respective data function size tables in ISO 20926 [117, Table A.2, page 23] as on page 262 in the appendix, $Party\ Plan$ contributes with 7 function points to the data function size of this requirements work package, i.e. $ILF_{size}(ILF_{complexity}, ILF) = (low, ILF) = 7\ function\ points$.

According to validation condition **V.xxii** and the respective data function size tables in ISO 20926 [117, Table A.2, page 23] as on page 262 in the appendix, $John\ and\ Lucy$ contributes with 0 function points to the data function size of this requirements work package, i.e. $EIF_{size} = 0\ function\ points$.


«Requirements Work Package» Party Plan Editing
PP!{Partymotto, Guest}
PE!{createPlan, registerGuest}
«lexicalDomain» Party Plan
$ILF_{Complexity} = \{low\}$
{guests invited to party}
«machine» Party editor
$data\ function$
«requirements» CE_FUR 01
JL!{plan, Partymotto,Guest}
$EIF_{Complexity} = \{n/a\}$
{plan a party }
«biddableDomain» John and Lucy

The validation conditions for this counting process activity are meet.

Activity 3. Determine Transactional Function

This activity serves to determine the amount of data movement, which is involved in the requirements work package.

The input to this activity requires a framed set of requirements, such as given by the requirements work package for “Party Plan Editing” and the transactional function complexity and sizes tables as defined in ISO 20926 and arranged on pages 262ff in the appendix of this work.

In contrast to the previous activities 1. to 2., which are concerned with determining the amount of data for the requirements work package in itself, the activity 3. allows for determining respective movements of this data that cross the application boundary, cf. figure 5.3 on page 43. It is about the interactions of the software with its environment, which is evaluated with regard to certain kinds of functionality, i.e. external input (EI), external inquiry (EQ), or external output (EO), which belong to the base functional components that are measurable according to ISO 20926 functional size measurement.

Six steps 3.a to 3.f have to be executed in order to identify and count relevant data movement at the application boundary.

At the end of this activity 3., the functional size for the transactional function of this requirements work package is calculated and given in function points.

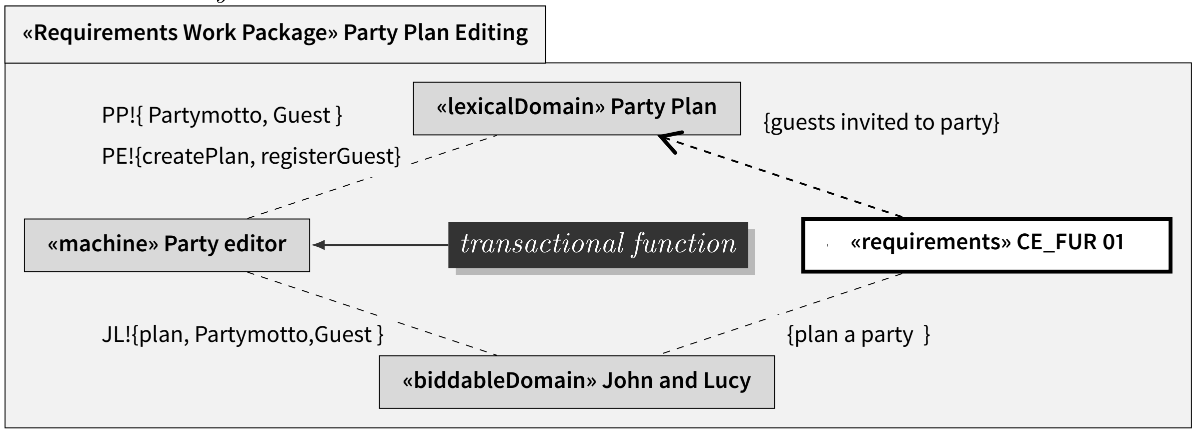
3.a Identify machine domain as transactional function

This step serves to locate the transactional function that manages data and control information, which crosses the application boundary. In accordance with step 2.b as discussed on page 90ff,it is already clear, which data is internal or external to the count, i.e. that problem domains serve as either an ILF or an EIF. Consequently, the machine domain with its interfaces to some ILF and EIF is evaluated as transactional function within a requirements work package.

There are no explicit validation conditions for this activity of the counting process.

EXAMPLE 6.8 Frame Counting Agenda – Activity 3./Step 3.a Determine transactional function

In the requirements work package for "Party Plan Editing" the problem domain $Party\ Plan$ is an ILF and $John\ and\ Lucy$ is an EIF.


«Requirements Work Package» Party Plan Editing
PP!{ Partymotto, Guest }
PE!{createPlan, registerGuest}
«lexicalDomain» Party Plan
{guests invited to party}
«machine» Party editor
transactional function
«requirements» CE_FUR 01
JL!{plan, Partymotto,Guest }
{plan a party }
«biddableDomain» John and Lucy

Due to the use of functional size measurement patterns to set up a requirements work package, each describes a problem at a basic level of detail, cf. table 5.3 on page 55, which maintains exactly one elementary process also known as transactional function that is to consider. In this example, it is the machine domain $Party\ editor$, which represents such a transactional function.

3.b Classify transactional function as either EI, EQ, or EO

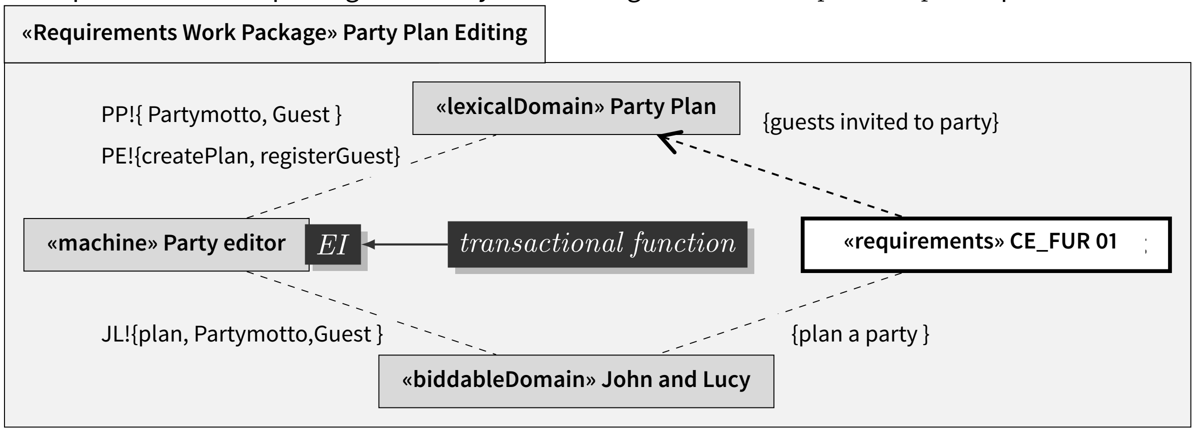
This step serves to decide on the counting rules and consequently the complexity and size tables that apply to evaluate a transactional function. It is determined by the type of functionality, i.e. the elementary process involved with a requirements work package and impacts the resulting function point value for its transactional function.

The following validation conditions safeguard this activity:

V.xxv The classification of the transactional function TF$_{Type}$::={EI|EQ|EO} is justified by the applied problem frame, and aligned with *Activity 1.*, validation condition **V.ii**.

EXAMPLE 6.9 Frame Counting Agenda – Step 3.b Classify transactional function

The requirements work package for "Party Plan Editing" fits to the *simple workpieces* problem frame.


«Requirements Work Package» Party Plan Editing
PP!{ Partymotto, Guest }
PE!{createPlan, registerGuest}
«lexicalDomain» Party Plan
{guests invited to party}
«machine» Party editor
EI
transactional function
«requirements» CE_FUR 01
JL!{plan, Partymotto,Guest }
{plan a party }
«biddableDomain» John and Lucy

According to the used functional size measurement pattern no. **#02**, **PF 2.7** taken from table 5.5 on page 72, the transactional function of this requirements work package belongs to an *external input* (EI) problem, such that $TF_{Type} = \{EI\}$. The validation condition **V.xxv** for this counting process activity holds.

3.c Count FTR for transactional function

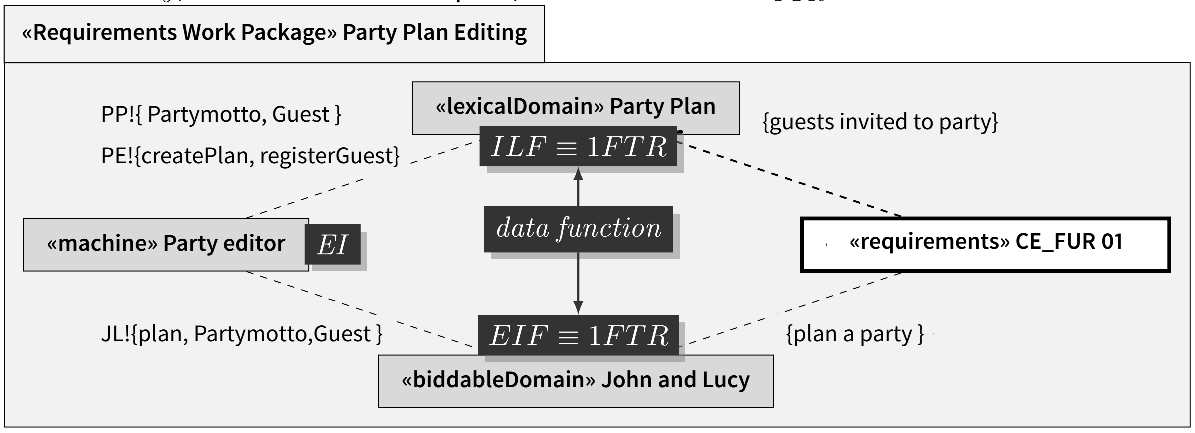
This step serves to identify the number of data functions or respective file types referenced (FTR), which are operated by the transactional functional within a requirements work package. It is to decide, how many interfaces need to be considered by the machine in order to establish the desired control, i.e. software functionality.

The following validation condition safeguards this activity:

V.xxvi The number of FTR counted for a TF corresponds to the number of DFs in this measurable problem: $TF_{FTR} = n$ ILF + m EIF.

EXAMPLE 6.10 Frame Counting Agenda – Step 3.c Count file types referenced

There are two data functions in this requirements work package, namely $Party\ Plan$ and $John\ and\ Lucy$, which conforms to step 2.a, with the result of $TF_{FTR} = 1\,ILF + 1\,EIF = 2\,FTR$.


«Requirements Work Package» Party Plan Editing
PP!{ Partymotto, Guest }
PE!{createPlan, registerGuest}
«lexicalDomain» Party Plan
{guests invited to party}
ILF ≡ 1FTR
«machine» Party editor
EI
data function
«requirements» CE_FUR 01
JL!{plan, Partymotto,Guest }
EIF ≡ 1FTR
{plan a party }
«biddableDomain» John and Lucy

The number of FTR counted for the TF of this requirements work package is two (2). Data movement at the machine interface to these FTR is of interest in the following. The validation condition V.xxvi for this counting process activity holds.

3.d Count DET for transactional function

This step serves to identify the number of data element types (DET) associated with a file type referenced (FTR) that cross the application boundary, i.e. information at the machine interface that is participating in a transactional function.

The following validation conditions safeguard this activity:

V.xxvii The number of DET counted for a TF corresponds to its shared, symbolic as well as causal phenomena.

V.xxviii Shared phenomena with a lexical domain do not cross the application boundary. They do not count in TF_{DET}.

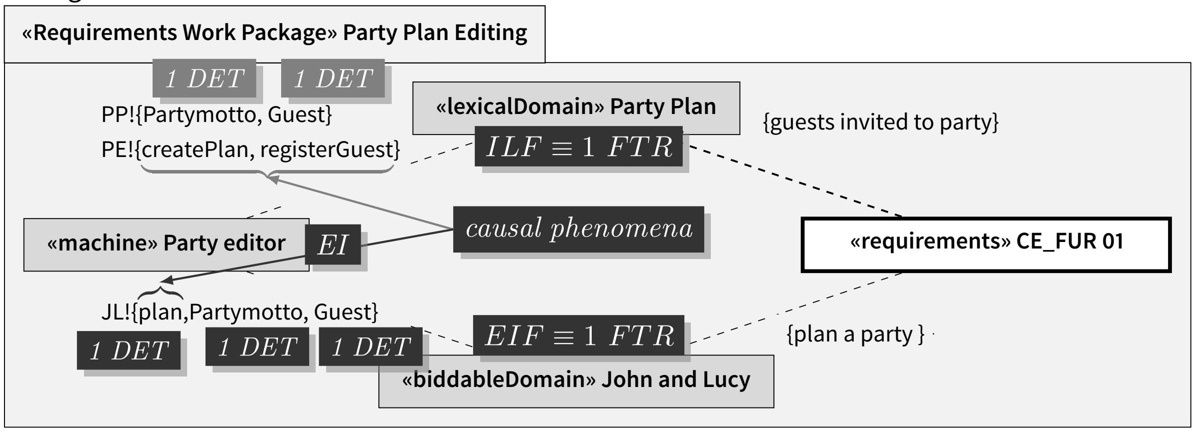
V.xxix Shared phenomena with a causal, biddable or display domain do cross the application boundary. They count in TF_{DET}.

V.xxx Each causal phenomenon that crosses the application boundary adds one (1) DET for the data element types of a transactional function TF_{DET}.

V.xxxi Each symbolic phenomenon that crosses the application boundary adds one (1) DET for the data element types of a transactional function TF_{DET}.

EXAMPLE 6.11 Frame Counting Agenda – Step 3.d Count data element types

According to validation condition V.xxvii the machine interface to $Party\ Plan$ and $John\ and\ Lucy$ must be taken into account. Since $Party\ Plan$ is a data function (ILF), which is of a lexical domain type (X), validation condition V.xxviii excludes its shared phenomena at the machine interface from the count. This kind of phenomena resides inside the application boundary and thus does not count for a transactional function. Following validation condition V.xxix to V.xxxi, the causal phenomenon $plan$ adds 1 DET to the transactional function, and the symbolic phenomena $PartyMotto$ and $Guest$ add together 2 DET.


«Requirements Work Package» Party Plan Editing
1 DET
1 DET
PP!{Partymotto, Guest}
PE!{createPlan, registerGuest}
«lexicalDomain» Party Plan
$ILF \equiv 1\ FTR$
{guests invited to party}
«machine» Party editor
EI
causal phenomena
«requirements» CE_FUR 01
JL!{plan,Partymotto, Guest}
1 DET
1 DET
1 DET
$EIF \equiv 1\ FTR$
{plan a party }
«biddableDomain» John and Lucy

The overall $TF_{DET} = 3\ DET$ for this requirements work package. The validation conditions for this counting process activity are meet.

3.e Determine functional complexity for transactional function

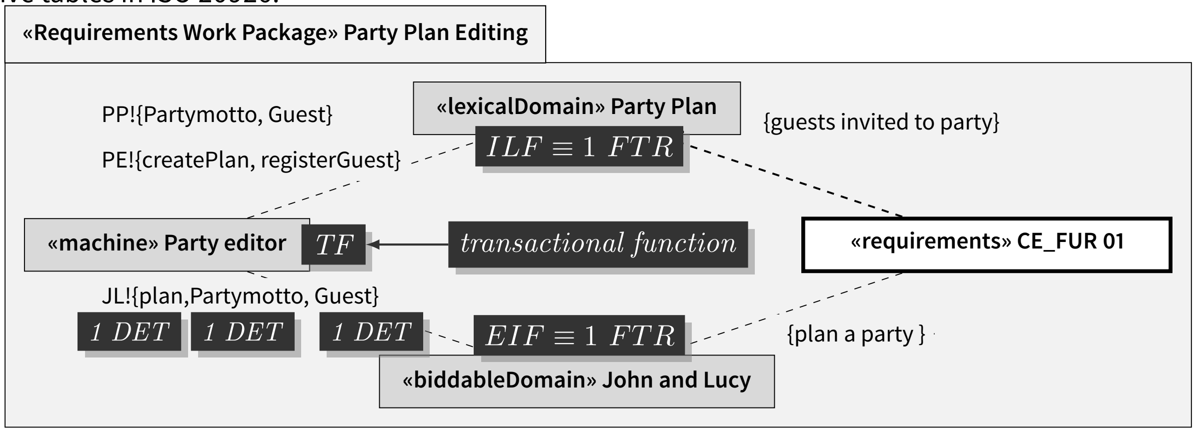
This step serves to determine the functional complexity of a requirements work package related to its transactional function. Its number of DET and related FTR is used as parameter to the ISO 20926 transactional function complexity tables Table A.3 and Table A.4 and results in a measure of the requirements complexity, which is either low, average, or high.

The following validation conditions safeguard this activity:

V.xxxii The TF functional complexity in this measurable problem is determined by $TF_{Complexity}(TF_{Type}, TF_{FTR}, TF_{DET})$::={low|average|high} according to the Table A.3 for EI, or Table A.4 for EO and EQ in ISO 20926.

EXAMPLE 6.12 Frame Counting Agenda – Step 3.e Determine transactional function complexity

In step 3.b the transactional function is classified with regard to its involved elementary process or with respect to its exhibited type of functionality, which is an external input $TF_{type} = \{EI\}$. According to step 3.c the number of file types referenced in this requirement work package is $TF_{FTR} = 2\ FTR$, which takes the problem domain $Party\ Plan$ and $John\ and\ Lucy$ into account. In step 3.d the number of data element types relevant to the transactional function is $TF_{DET} = 3\ DET$, since only the shared phenomena at the machine interface with $John\ and\ Lucy$ are to consider. Consequently, the functional complexity for the transactional function in this requirements work package is $TF_{Complexity}(TF_{Type}, TF_{FTR}, TF_{DET}) = TF_{Complexity}(EI, 2, 3) = \{low\}$ following the respective tables in ISO 20926.


«Requirements Work Package» Party Plan Editing
PP!{Partymotto, Guest}
PE!{createPlan, registerGuest}
«lexicalDomain» Party Plan
ILF ≡ 1 FTR
{guests invited to party}
«machine» Party editor
TF
transactional function
«requirements» CE_FUR 01
JL!{plan,Partymotto, Guest}
1 DET
1 DET
1 DET
EIF ≡ 1 FTR
{plan a party }
«biddableDomain» John and Lucy

The validation condition **V.xxxii** for this counting process activity is met.

3.f Determine functional size for transactional function

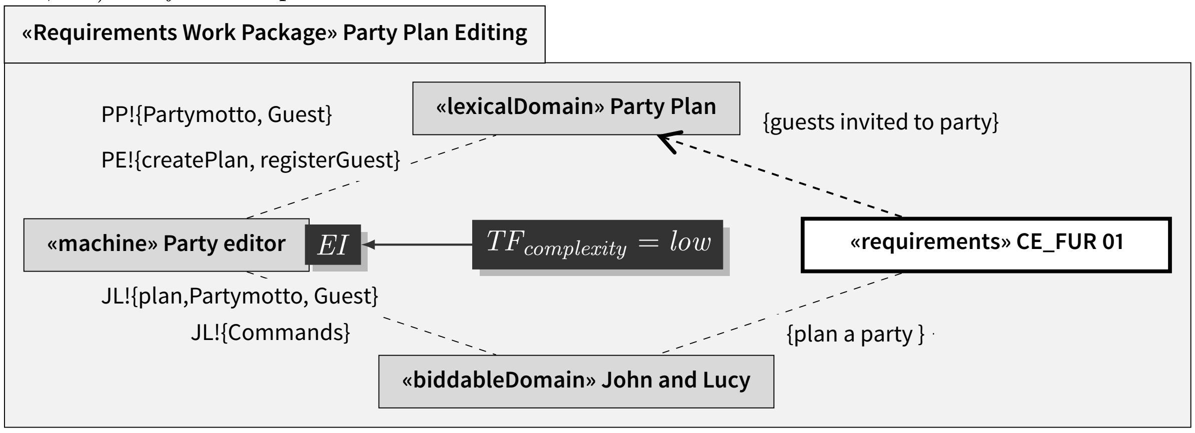
This step serves to determine the functional size of a requirements work package related to its transactional function. Its functional complexity is used as parameter to the ISO 20926 transactional function size table and results in a measure given in function points between 3 and 7.

The following validation conditions safeguard this activity:

V.xxxiii The TF functional size in this measurable problem is given in function points and determined by $TF_{Size}(TF_{Complexity}, TF_{Type})$ according to Table A.5 in ISO 20926.

EXAMPLE 6.13 Frame Counting Agenda – Step 3.f Determine transactional function size

The transactional function *Party editor*, which is according to the previous step 3.c an *external input* of *low* complexity, has a functional size of $TF_{Size}(TF_{Complexity}, TF_{Type}) = (low, EI) = 3\,function\ points$.


«Requirements Work Package» Party Plan Editing
PP!{Partymotto, Guest}
«lexicalDomain» Party Plan
{guests invited to party}
PE!{createPlan, registerGuest}
«machine» Party editor
EI
$TF_{complexity} = low$
«requirements» CE_FUR 01
JL!{plan,Partymotto, Guest}
JL!{Commands}
{plan a party }
«biddableDomain» John and Lucy

The validation condition V.xxxiii for this counting process activity is met.

Activity 4. Report Functional Size for FUR

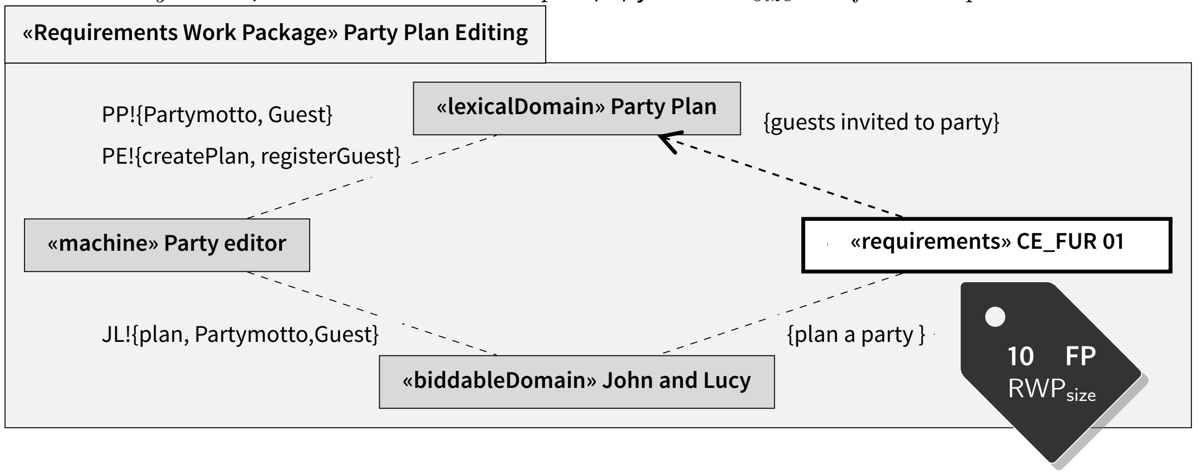
This activity serves to conclude the requirements sizing method for a requirements work package. So far, by means of functional size measurement patterns, a set of requirements is fit to a measurable problem. Its involved base functional components, namely its data and transactional functions, provides for determining respective function point values. Now, The functional size of this requirements work package is compiled by these point values.

The following validation condition safeguards this activity:

V.xxxiv The functional size for a measurable problem is reported in function points (FP), following the format: <Measurable Problem$_{size}$> FP (IFPUG-ISO/IEC 20926:2009-FCA), which indicates by the postfix of –FCA (for Frame Counting Agenda) a customization of the IFPUG standard. The functional size of a measurable problem is a cumulated value of DF and TF sizes given in function points:
$Measurable\ Problem_{size} = ILF_{Size} + EIF_{Size} + TF_{Size}$.

EXAMPLE 6.14 Frame Counting Agenda – Activity 4. Report the requirement's functional size

The requirements work package "Party Plan Editing" implements the requirement **CE_FUR 01**. It fits to the *simple workpieces* problem class, which is a functional size measurement pattern and thus a measurable problem. There are two data functions and one transactional function that can be measured for this requirements set. According to step 2.f there is only one ILF *Party Plan* which yields $ILF_{size} = 7\ function\ points$. In addition, there is only one EIF *John and Lucy*, which due to some measurement rules yields $EIF_{size} = 0\ function\ points$. According to step 3.f the transactional function *Party editor*, which is an *external input* (EI) yields $TF_{size} = 3\ function\ points$.


«Requirements Work Package» Party Plan Editing
PP!{Partymotto, Guest}
PE!{createPlan, registerGuest}
«lexicalDomain» Party Plan
{guests invited to party}
«machine» Party editor
«requirements» CE_FUR 01
JL!{plan, Partymotto,Guest}
{plan a party }
«biddableDomain» John and Lucy
10 FP
RWP size

The overall functional size for this requirements work package is $Measurable\ Problem_{size} = ILF_{Size} + EIF_{Size} + TF_{Size} = 7 + 0 + 3 =$ **10 FP (IFPUG-ISO/IEC 20926:2009-FCA)**.
The validation condition V.xxxiv for this counting process activity is met.

Completing activity 4 of the frame counting agenda, produces the final output of the requirements sizing method:

- a set of requirements grouped to a requirements work package
  by means of functional size measurement patterns. These patterns have helped to identify the relevant base functional components that undergo the counting process, and
- whose functional size is determined consistently
  safeguarded by validation conditions, which are integrated to the requirements sizing method. These validation conditions maintain the rules of the counting game and thus care for reproducible requirements' size measures.

The frame counting procedure is repeated for remaining requirement statements (FUR), respectively.

6.5. Sample Application to Jackson's Basic Frames

In this section 6.5, the Frame Counting Agenda as introduced in section 6.3.1 is demonstrated by making use of some problems that base on well-established frames from Jackson [128]. Other case examples, such as for a well-elaborated Vacation Rentals Web Application in chapter 10, are part of considerations in Part V, which presents different Case Studies. These sample applications recapitulate each activity of the requirements sizing method. They illustrate how a functional size measurement pattern determines problem count and composition for a set of requirements, and with it the problem's functional size measure.

The choice of sample applications to the requirements sizing method discussed in this section is done in relation to their involved type of functionality, which distinguishes different classes of measurable problems as summarized in table 5.5 on page 72.

The Party Plan Editing problem in section 6.5.1 is an instance of the simple workpieces frame. It is measured according to the rules of an external input, since it is concerned with processing some information that is received (TOFF-i).

The Local Traffic Monitoring problem in section 6.5.2 is an instance of the information display frame. It is measured according to the rules of an external inquiry, since it is concerned with processing some information that is retrieved (TOFF-ii).

The Occasional sluice gate in section 6.5.3 is an instance of the commanded behaviour frame in section 6.5.3. It is measured according to the rules of an external output, since it is concerned with processing some information that is derived (TOFF-iii).

For each of these different measurement problems it is discussed, how the problem count is executed, i.e. how the problem's functional size is determined based on the respective functional size measurement patterns.

6.5.1. Counting a Simple Workpieces Problem: Party Plan Editing

All the details, regarding the execution of the counting procedure by means of Jackson [128, pages 125–129] "simple workpieces" frame used as functional size measurement pattern, are discussed in section 6.4 Step-By-Step Guide to the Requirements Sizing Method.

6.5.2. Counting an Information Display Problem: Local Traffic Monitoring

The problem diagram for local traffic monitoring [128, Page 95] is an instance of the problem frame "information display" [128, Page 93]. Table 5.5 lists this frame by **#07 PF 2.6** as a functional size measurement pattern.

The monitoring computer retrieves information from some sensors, that indicate vehicles passing the street. To satisfy the requirements statement, this information is processed to a report, which is written as output to a strip printer.

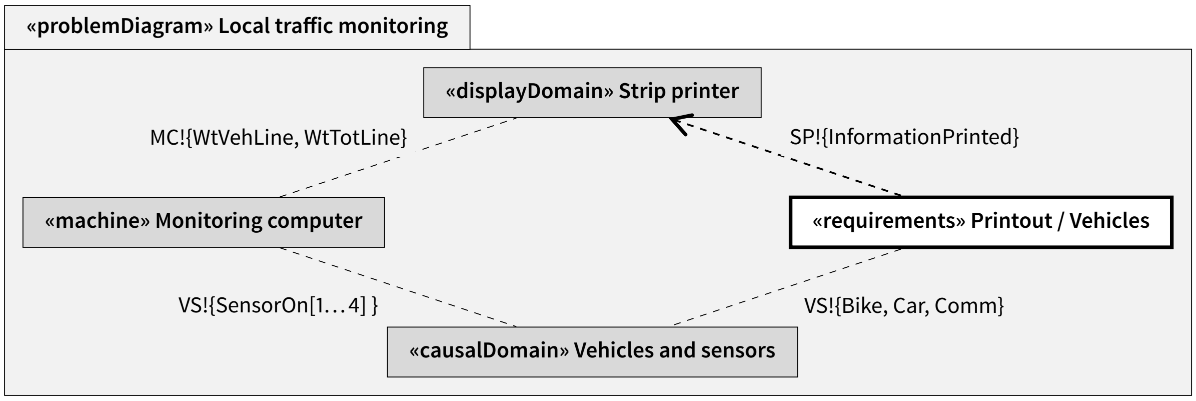

«problemDiagram» Local traffic monitoring
«displayDomain» Strip printer
MC!{WtVehLine, WtTotLine}
SP!{InformationPrinted}
«machine» Monitoring computer
«requirements» Printout / Vehicles
VS!{SensorOn[1…4] }
VS!{Bike, Car, Comm}
«causalDomain» Vehicles and sensors

TABLE 6.4 Local traffic monitoring is a measurable information display problem

Comments on counting process activity	Results of activity
1. Classify FUR by Functional Size Measurement Patterns.	
Local Traffic Monitoring is a measurable problem, because it fits the *information display* FSM pattern #07 in table 5.5. Applied validation conditions: **V.i - V.iii**	information display problem
2. Determine Data Functions.	
Local Traffic Monitoring has two problem domains, namely the causal domain *Vehicles and sensors* and the display domain *Strip printer*.	
2.a Identify problem domains as data functions.	
The domain *Vehicles and sensors* shares only causal phenomena (events) at the machine interface, it is no data function. The domain *Strip printer* shares symbolic phenomena at the machine interface. It is a data function. Applied validation conditions: **V.iv**, **V.v**	1 data function: strip printer
2.b Classify data functions into ILF or EIF.	
The only data function in this measurable problem *Strip printer* is an internal logical file, because it is constrained. Applied validation conditions: **V.vi**, **V.x**	1 ILF: strip printer
2.c Count DET for each data function.	
The data function *Strip printer* shares two symbolic phenomena at the machine interface, namely *WtVehLine* and *WtTotLine*, which represent the data element types of this data function. Applied validation conditions: **V.xi**	$DET_{strip\ printer} = 2$
2.d Count RET for each data function.	
There is only one data function *Strip printer*, which corresponds to 1 RET. Applied validation conditions: **V.xii**	$RET_{strip\ printer} = 1$
2.e Determine functional complexity for data functions.	
Strip printer is the only data function, i.e. an ILF in this measurable problem, which has 2 DET according to step 2.c and represents 1 RET according to step 2.d. Respectively, $ILF_{DET} = \sum_{i=1}^{n} DET_{ILF_i} = DET_{strip\ printer}$ and $ILF_{RET} = \sum_{i=1}^{n} RET_{ILF_i} = RET_{strip\ printer}$.	$ILF_{DET} = 2$ $ILF_{RET} = 1$

Comments on counting process activity	Results of activity
Table A.1 of ISO 20926 given in the appendix on page 262 is used to determine the respective data function complexity $ILF_{Complexity}(ILF_{RET}, ILF_{DET})$ by means of these RET and DET values, that is $ILF_{Complexity}(1, 2) = low$. Applied validation conditions: **V.xiii**, **V.xvii**, **V.xx**	$ILF_{Complexity} = low$
2.f Determine functional size for data functions.	
Table A.2 of ISO 20926 given in the appendix on page 262 is applied to determine the respective data function size $ILF_{Size}(ILF_{Complexity}, ILF)$ using its data function complexity determined in the previous step 2.e., that is $ILF_{Size}(low, ILF) = 7$. Applied validation conditions: **V.xxiii**	$ILF_{Size} =$ 7 function points
3. Determine Transactional Function.	
Local traffic monitoring has one machine domain $Monitoring\ computer$.	
3.a Identify machine domain as transactional function.	
The machine domain $Monitoring\ computer$ represents the transactional function in this measurable problem.	1 transactional function: monitoring computer
3.b Classify transactional function as either EI, EQ, or EO.	
In accordance with activity 1., this measurable problem fits an information display frame, which is a functional size measurement pattern as defined in table 5.5 for determining the functional size of an external inquiry (EQ). Applied validation conditions: **V.xxv**	$TF_{type} = EQ$
3.c Count FTR for transactional function.	
$Monitoring\ computer$ involves only one machine interface to a data function as defined in step 2.a, which is the file type referenced to consider in this step. One data function and no external interface file results in $n = 1$ and $m = 0$ for TF_{FTR} = n ILF + m EIF = 1 + 0 = 1. Applied validation conditions: **V.xxvi**	$TF_{FTR} = 1$
3.d Count DET for transactional function.	
At the machine interface of $Monitoring\ computer$ are two symbolic phenomena shared with $Strip\ printer$ and four causal phenomena shared with $Vehicles\ and\ sensors$. Each of these six phenomena crosses the application boundary and thus counts in the transactional function as DET. Applied validation conditions: **V.xxvii**, **V.xxx**, **V.xxxi**	$TF_{DET} = 6$
3.e Determine functional complexity for transactional function.	
Table A.4 for EQ of ISO 20926 given in the appendix on page 262 is used to determine the transactional function complexity $TF_{Complexity}(TF_{Type}, TF_{FTR}, TF_{DET}) = TF_{Complexity}(EQ, 1, 6)$ of $Monitoring\ computer$ by means of the FTR and DET values from step 3.c and 3.d. Applied validation conditions: **V.xxxii**	$TF_{Complexity} = low$
3.f Determine functional size for transactional function.	
Table A.5 of ISO 20926 given in the appendix on page 262 is applied to determine the respective transactional function size of $Monitoring\ computer$ $TF_{Size}(TF_{Complexity}, TF_{Type}) = TF_{Size}(low, EQ)$ by using its transactional function complexity determined in the previous step 3.e. Applied validation conditions: **V.xxxiii**	$TF_{Size} =$ 3 function points
4. Report Functional Size for FUR.	
Data and transactional function size for this measurable problem are determined by $MeasurableProblem_{size} = ILF_{Size} + EIF_{Size} + TF_{Size} = 7 + 0 + 3 = 10$. Applied validation conditions: **V.xxxiv**	$LocalTraffic-Monitoring_{size} =$ **10 function points (IFPUG-ISO/IEC 20926:2009-FCA)**

6.5.3. Counting a Commanded Behaviour Problem: Occasional Sluice Gate

The problem diagram for an occasional sluice gate [128, Page 91] is an instance of the problem frame "commanded behaviour" [128, Page 89]. Table 5.5 lists this frame by **#14 PF 2.8** as a functional size measurement pattern.

To satisfy the requirements statement, the sluice controller derives out of varying information from the gate state and operator commands, how to control the gate motor.

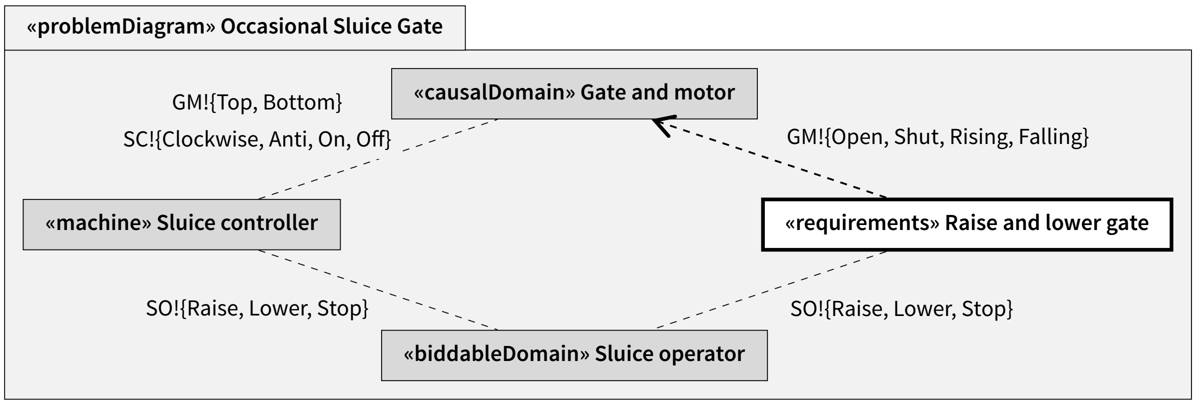

«problemDiagram» Occasional Sluice Gate
«causalDomain» Gate and motor
GM!{Top, Bottom}
SC!{Clockwise, Anti, On, Off}
GM!{Open, Shut, Rising, Falling}
«machine» Sluice controller
«requirements» Raise and lower gate
SO!{Raise, Lower, Stop}
SO!{Raise, Lower, Stop}
«biddableDomain» Sluice operator

TABLE 6.6 Occasional sluice gate is a measurable commanded behaviour problem

Comments on counting process activity	Results of activity
1. Classify FUR by Functional Size Measurement Patterns.	
Occasional Sluice Gate is a measurable problem, because it fits the *commanded behavior* FSM pattern #14 in table 5.5. Applied validation conditions: **V.i - V.iii**	commanded behavior problem
2. Determine Data Functions.	
Occasional Sluice Gate has two problem domains, namely the causal domain *Gate and motor* and the biddable domain *Sluice operator*.	
2.a Identify problem domains as data functions.	
The domain *Gate and motor* shares state information , which can be interpreted as symbolic phenomena, at the machine interface. It can be classified as data function, when following the discussion in Jackson [128, pages 83, 84]. The domain *Sluice operator* shares no symbolic phenomena at the machine interface. It is no data function, respectively. Applied validation conditions: **V.iv**, **V.v**	1 data function: gate and motor
2.b Classify data functions into ILF or EIF.	
The only data function in this measurable problem *Gate and motor* is an internal logical file. Applied validation conditions: **V.vi**, **V.viii**	1 ILF: gate and motor
2.c Count DET for each data function.	
The data function *Gate and motor* shares two symbolic phenomena at the machine interface, namely *Top* and *Bottom*, which represent the data element types of this data function. Applied validation conditions: **V.xi**	$DET_{gate\ and\ motor} = 2$
2.d Count RET for each data function.	
There is only one data function *Gate and motor*, which corresponds to 1 RET. Applied validation conditions: **V.xii**	$RET_{gate\ and\ motor} = 1$
2.e Determine functional complexity for data functions.	
Gate and motor is the only data function, i.e. an ILF in this measurable problem, which has 2 DET according to step 2.c and represents 1 RET according to step 2.d. Respectively, $ILF_{DET} = \sum_{i=1}^{n} DET_{ILF_i} = DET_{gate\ and\ motor}$ and $ILF_{RET} = \sum_{i=1}^{n} RET_{ILF_i} = RET_{gate\ and\ motor}$.	$ILF_{DET} = 2$ $ILF_{RET} = 1$

Comments on counting process activity	Results of activity
Table A.1 of ISO 20926 given in the appendix on page 262 is used to determine the respective data function complexity $ILF_{Complexity}(ILF_{RET}, ILF_{DET})$ by means of these RET and DET values, that is $ILF_{Complexity}(1,2) = low$. Applied validation conditions: **V.xiii**, **V.xvii**, **V.xx**	$ILF_{Complexity} = low$
2.f Determine functional size for data functions.	
Table A.2 of ISO 20926 given in the appendix on page 262 is applied to determine the respective data function size $ILF_{Size}(ILF_{Complexity}, ILF)$ using its data function complexity determined in the previous step 2.e., that is $ILF_{Size}(low, ILF) = 7$. Applied validation conditions: **V.xxiii**	$ILF_{Size} =$ 7 function points
3. Determine Transactional Function.	
Occasional Sluice Gate has one machine domain *Sluice controller*.	
3.a Identify machine domain as transactional function.	
The machine domain *Sluice controller* represents the transactional function in this measurable problem.	1 transactional function: sluice controller
3.b Classify transactional function as either EI, EQ, or EO.	
In accordance with activity 1., this measurable problem fits a commanded behavior frame, which is a functional size measurement pattern as defined in table 5.5 for determining the functional size of an external output (EO). Applied validation conditions: **V.xxv**	$TF_{type} = EO$
3.c Count FTR for transactional function.	
Sluice controller involves only one machine interface to a data function as defined in step 2.a, which is the file type referenced to consider in this step. One data function and no external interface file results in $n = 1$ and $m = 0$ for TF_{FTR} = n ILF + m EIF = 1 + 0 = 1. Applied validation conditions: **V.xxvi**	$TF_{FTR} = 1$
3.d Count DET for transactional function.	
At the machine interface of *Sluice controller* are two symbolic phenomena and four causal shared with *Gate and motor*, and the machine shares three causal phenomena with the *Sluice operator*. Each of these nine phenomena crosses the application boundary and thus counts in the transactional function as DET. Applied validation conditions: **V.xxvii**, **V.xxix**, **V.xxx**, **V.xxxi**	$TF_{DET} = 9$
3.e Determine functional complexity for transactional function.	
Table A.4 for EO of ISO 20926 given in the appendix on page 262 is used to determine the transactional function complexity $TF_{Complexity}(TF_{Type}, TF_{FTR}, TF_{DET}) = TF_{Complexity}(EO, 1, 9)$ of *Sluice operator* by means of the FTR and DET values from step 3.c and 3.d. Applied validation conditions: **V.xxxii**	$TF_{Complexity} = low$
3.f Determine functional size for transactional function.	
Table A.5 of ISO 20926 given in the appendix on page 262 is applied to determine the respective transactional function size of *Sluice operator* $TF_{Size}(TF_{Complexity}, TF_{Type}) = TF_{Size}(low, EO)$ by using its transactional function complexity determined in the previous step 3.e. Applied validation conditions: **V.xxxiii**	$TF_{Size} =$ 4 function points
4. Report Functional Size for FUR.	
The size for this measurable problem is determined by $MeasurableProblem_{size} = ILF_{Size} + EIF_{Size} + TF_{Size} = 7 + 0 + 4.$ Applied validation conditions: **V.xxxiv**	$Occasional-SluiceGate_{size} =$ **11 function points (IFPUG-ISO/IEC 20926:2009-FCA)**

6.6. Discussion & Related Work

The problem-based estimating method presented in this chapter relies on the use of requirement patterns to keep the counting procedure synchronized with the requirements model that is represented by problem diagrams. This is very beneficial in the context of unavoidable requirements change.

It allows for monitoring changes to size estimates on-the-fly [28, page 5] in the presence of tool support, which also contributes to requirements understanding and negotiation.

Such a kind of tool support is within reach. Problem-based estimating as introduced here is realizable as an extension of the UML4PF eclipse plugin [107], which assists requirements modeling that builds on problem frames and that could make automating early-phase size measures possible.

The International Function Point Users Group (IFPUG), who maintains ISO/IEC 20926:2009 functional size measurement standard, offers certification of software that provides for function point counting. In order to evaluate, if an extension of the UML4PF plugin by the pattern-led, problem-based functional size measurement method as proposed in this dissertation is reasonably possible, the respective criteria for Certification of Function Point Software type 2 [121] have been considered for reviewing the interpretation of ISO 20926 as defined in the frame counting agenda and its validation conditions given in this chapter.

The result of this review is promising. All certification criteria that require automated identification by the software are fulfilled by the proposed requirements sizing method. Even though two issues arise, which are resolvable as discussed in appendix B.2.

Functional size measurement in the context of embedded and real-time systems lead to specialized counting regimes such as Full Function Points [1] that belong to the COSMIC universe [142]. These systems are not objects of investigation in this work.

Functional size measurement according to the IFPUG standard given in ISO 20926 is more adapted for information systems. This becomes obvious with regard to the focus on functional user requirements (FUR) that are considered by the IFPUG standard, and the comparatively low function point values that can be obtained by measurable problems with a high amount of control information, which becomes apparent by the sample measurement of Jackson's occasional sluice gate in section 6.5.3.

This work does not challenge the suitability of any point value that is determined for a measurable problem. For instance, it does not question if 11 function points represents a proper size for Jackson's occasional sluice gate problem or not. It does not strive towards improving any measurement standard in itself. It strives towards improving the applicability of functional size measurement by integrating it to a requirements engineering approach.

The contribution as proposed in this chapter is a reproducible requirements estimate, one which is represented by consistent point values. This is the intended use of problem-based estimating within an agile project management process and beyond. It establishes a key "to meet expectations more consistently" [221, page 161], i.e. by qualifying teams for taking credible decisions at an early stage in the software project.

6.7. Summary

This chapter gives the details on a method for determining function points for requirements in a reproducible way: its procedure, its step-by-step execution, and its application to several examples.

This requirements sizing method is documented in the frame counting agenda given in table 6.2 on page 82, and builds on problem-based functional size measurement patterns. It makes reproducible size measures for requirements possible, since each step of the counting procedure follows the measurement process defined in ISO/IEC 20926:2009 [117, section 5, pages 8-19, and 21] and each step is customized to work on the constituent parts of measurable problems.

Validation conditions given in table 6.3 on page 86 accompany each step during the counting procedure. They are at the heart of the measurement process and provide interpretations of the counting rules expressed in ISO 20926. They care for a transparent requirements sizing process and thereby safeguard the function points that apply to a requirements work package against inconsistency.

The problem-based functional size measurement approach as introduced in this chapter mitigates major sources of difficulties that are the root cause of wrong counts, and which are involved in the fact that function point "counting requires a considerable amount of interpretation [...which] usually fail to keep synchronized with the requirements" [28, page 1].

Problem-based functional size measurement patterns establish a proxy [114] that meets this concern. They serve the decomposition of requirements to measurable problems, whose constituent parts relevant for the counting process are recognizable and during the measurement process subject to interpretation by defined validation conditions.

Omission and duplication are frequent errors in requirements estimating, which often originates from a malpositioned application boundary [155], one that neglects to feature requirements dependencies [28, page 4]. This gives reason for the observation that determining the respective measurable components of requirements is observed as the most critical and error-prone step at the same time [104, page 205] in industrial practice [120]. The proposed requirements sizing method explicitly addresses this issue. Each requirements work package represents a unit of measure for requirements, which makes explicit allowance for requirements dependencies. They unite desired software functionality, such that it is coherent inside and among measurable problems. As introduced and demonstrated by this work, this adds a new perspective on problem count and composition, which contributes to stabilized requirements estimates.

Part III.

Problem-Based Project Adaptation

*Part III Problem-Based Project Adaptation is about answering **RQ 2** How to adjust speed? It develops the conceptions and methods for determining 'instant options for action' namely design alternatives, which provide the project team with a credible route of development work to be "done" within the project timebox, for satisfying a defined scope of software product requirements. Since Problem-Based Project Adaptation relies entirely on pattern practices, decisions made and development options planned for producing desired project deliverables can be replayed and revised by the team as needed to impact their software project speed. Chapter 7 Problem-Based Units of Work refines the concept of transition schemas known from structured analysis to Transition Templates, for establishing a link between patterns of software problem analysis and those of software solution design. Transition Templates assemble instant models for exploring design alternatives that fit a defined scope of software product requirements, such as given by Requirements Work Packages. Chapter 8 Problem-Based Adaptation Framework develops the 4+1 view model on software architecture further, such that it benefits from Transition Templates, as introduced in the previous chapter 7, and operates on patterns exclusively. The resulting "One4All" view model on software architecture guides the project team in establishing units of work, which provides them with a blueprint or plan for the anticipated delivery of desired working software. On the basis of patterns, the "One4All" view model stabilizes the leeway for the fulfillment of Requirements Work Packages. It eases the integration of recurring development problems to predetermined technology platforms, and supports work plan prioritization according to the software product life-cycle and its projectable value delivery. The enhanced anticipation of development plans and the improved adaptability of requirements fulfillment are two cruicial points of controlling project speed.*

7. Problem-Based Units of Work

7.1. Introduction

This chapter is about identifying best practices for a requirements work package, that guide the team on its way to develop desired outcome.

A meet-in-the-middle approach between emerging software problems on the one hand and an enduring solution design on the other hand, which is ultimately determined by the (intended) software platform in-use, is elaborated for establishing reasonable foresight. This approach results in a problem-based unit of work, which constitutes an Architectural Blueprint (ABP) of what "done" looks like, i.e. a plan to satisfy user expectations, and how to work it out.

Section 7.2 Background gives a brief introduction to what transition schemas are, and motivates the use of patterns in software architecture design. These concepts are combined in this chapter for creating on the fly documentation, which enables a smooth and traceable hand-over of user requirements to software development.

Section 7.3 Transition Templates – Making problems absorb into platform introduces a new kind of patterns named "transition templates", that pave the way for creating any architectural blueprint for a given requirements work package, since problem and solution model can now be ascribed to a common pattern base of best practices. In addition, these templates help to maintain a constant level of detail for describing problem scenarios and accordant problem compositions involved with requirements work packages. This property of transition templates is of great use for keeping track on requirement dependencies, and which is taken advantage of in the subsequent chapter 8 Problem-Based Adaptation Framework for arranging the software life cycle and for the prioritization of requirements.

Section 7.4 Problem templates applies the newly formed concept of a transition template to the domain of software analysis, i.e. to problem-based functional size measurement patterns. This adds a dynamic perspective on a problem covered by a requirements work package.

Section 7.5 Solution templates exemplifies the application of transition templates to the domain of software design. It not only elaborates a procedure for turning design patterns common in the field of software architecture into solution templates, it also develops a classification for separating patterns of fine-grained software design from those which have significance in coarse-grained software architectures.

Section 7.6 Discussion & Related Work discusses the findings of this chapter in the light of a size-driven and pattern-based software project process. It considers relevant related work and addresses the question of how much planning is enough. It argues that setting up a problem-based unit of work by means of transition templates is one possible answer to it.

Section 7.7 Summary summaries the need and contribution of an integrated view to problem and respective solution models as provided by a problem-based unit of work. This integrated view does not only motivate planned functionality by making its architectural blueprint and respective decision-making explicit. It also allows for establishing and comparing alternative plans for the development of desired outcome, which is guided by transition templates.

7.2. Background

This section presents state-of-the-art literature and concepts used in the following for developing problem-based units of work. These are for describing an engineering plan (architectural blueprint) for the development of a software design created by means of patterns, which is applicable for solving the problem covered in a requirements work package.

That way, project teams get in the position to early anticipate and organize their upcoming work, and at the same time taking advantage of reusing commonly known best practices, which have already demonstrated their effectiveness.

This section focuses on the patterns of solution design, their composition, and dependency to problem analysis, for making a smooth, pattern-guided, and model-based transition between these two domains possible.

Section 7.2.1 Architectural Blueprints and Pattern-Oriented Analysis and Design gives an overview of what is considered a pattern in software design, the challenges involved with their application, and how Yacoub and Ammar [230] master these by POAD in a model-driven way.

Section 7.2.2 Transformation Schemas explains how Ward and Mellor approach the allocation of problems to a candidate solution (architecture), i.e. how to map desired software functionality (tasks) to those computational components (processors), which can process and thus fulfill these.

Their concept of transformation schemata is adapted in this chapter to establish a constant level of abstraction for the application of commonly known patterns, which helps the members of a project team in bridging the semantic gap, that distracts them from a shared understanding of the problem, and from their operative collaboration in finding its practicable solution.

7.2.1. Architectural Blueprints and Pattern-Oriented Analysis and Design

As research[1] by Bosch confirms, “80-90% of all R&D effort [and so its related software development projects] is allocated to commodity functionality[, ...] this functionality needs to work, but does not help the company distinguishing itself from its competitors and through that drive sales. [...It is] functionality that no customer cares about. It just has to work and, if it does, nobody cares.” [40]. Consequently, Ambler stated that project planning has to “accept that some architectural decisions [especially against the background of commodity functionality] are already made” [12, page 281].

Corresponding to that, Yacoub and Ammar’s summary is not suprising, that “To improve software productivity, we have to stop developing applications from scratch and make use of existing solutions that have been applied, tested and proven useful in successful projects. Patterns[2] promise new design reuse benefits early in the development lifecycle.” [229] To this end, they propose a pattern-oriented analysis and design approach (POAD), which operates on UML models, to ease the integration of exisiting design patterns into a combined solution.

Patterns of software design are “descriptions of and solutions to recurring problems” [137, page 39]. They have become popular by the work of Gamma et al. [90] (1995) know as the “Gang of Four”, who took the idea of a pattern language for the development of "architectural blueprints" from the architect Alexander et al. [5] (1978) to the domain of software engineering, for describing recurring forms of solving complex engineering tasks in a structured and reusable, since model-based way. Since then, many pattern catalogues have evolved, of which chapter D in the appendix VII provides a brief list.

As Yacoub and Ammar rightly recognize, “Much work is expended in discovering patterns in various domains. Techniques[, which define a pattern composition approach] to deploy these proven design solutions are still lacking systematic support. [Developing these] to glue patterns, facilitate design learning as composition of patterns, maintain pattern level review of design, and solve traceability problems.” [229], their POAD approach is entirely dedicated to patterns of early software design. It does not include patterns of problem analysis.

The challenge addressed in the following is to establish a pattern composition approach, which allows for the co-development of requirements and software architecture; one which enables integrated requirements engineering as formulated by Sommerville [201, 202], and thus to bridge the gap between requirements and software architecture as illustrated by Nuseibeh’s twin peaks model [162].

As Larsen stated, “Designing systems using components and proven solutions [(patterns)] elevates the abstraction at which engineers work[, ...] communicate and produce solutions. Productivity [due to larger abstractions] and quality [due to component reuse] are the two main drivers this approach brings.” [137, pages 38 and 39]. Identifying a proper level of abstraction, one which accounts for “the packaging [...and thus] the pathways for making reuse possible” [137, page 39], is a key [156, page 112] to benefit from pattern practices, and to make the allocation of problems to suitable solutions possible through these.

[1]The three-layer product model as developed by Bosch classifies software functionality into either innovative, differentiating or commodity [38, page 37, figure 1]

[2]as well as frameworks [130], which “help to create reusable, approachable software architectures” [137, page 38]

7.2.2. Transformation Schemas

Ward and Mellor (1985) developed transformation schemas, which model "a system as an active entity – as a network of activities that accept and produce data and control messages [. This concept . . .] is based on the notation for data flow diagrams proposed by DeMarco [74]" [217, page 41] for structured analysis [231], and extend this notation by means for modeling and executing the timely order of activities with improved rigour. That way, Ward and Mellor contribute to the domain of real-time systems engineering.

As illustrated on the left side in figure 7.1, a transformation schema represents a "cluster of transformations" [217, page 43], which captures a combination of related *data transformations* (solid circle) and *control transformations* (dashed circle), and gives a name to these grouped activities. The number of transformations covered by one schema is not limited per definition, that is, the respective complexity regarding instances and interfaces of transformations in a transformation schema depends on the experience and practices used by the modeler. Ward and Mellor [219] provide heuristics for structuring the modeling process, e.g. they give rules for enumerating transformations, and a leveling strategy (top-down approach) for cascading the details of each.

A **transformation schema** accepts control (events) and information (data), which are processed by respective control and data transformations. Within a transformation schema a control transformation triggers data transformations and vice versa. Storage capabilities (boxes) exists for both, events and data, which enables persistence and synchronization of these.

Each **control transformation** "maps input event flows to output event flows" [217, page 64]. Its behaviour is describeable by state-transition diagrams [217, page 65], which represent a finite automaton with output [. . . such as organized by] a Mealy machine [, and which] associates each transition (that is, each combination of state and input) with an output symbol" [217, pages 67, 68].

A **data transformation** "represents the work done to produce the outputs" [217, page 41]. There "is a continuum of possible specification techniques" [217, page 82] for it, where a procedural description, which produces "problem-oriented[, . . .] sets of rules for calculating the values of the outputs of a transformation if the values of the inputs is provided." [217, page 82] "The precondition for creating a specification for a data transformation is that the data used[, stored] and produced by the transformation be completely specified", too [217, page 81].

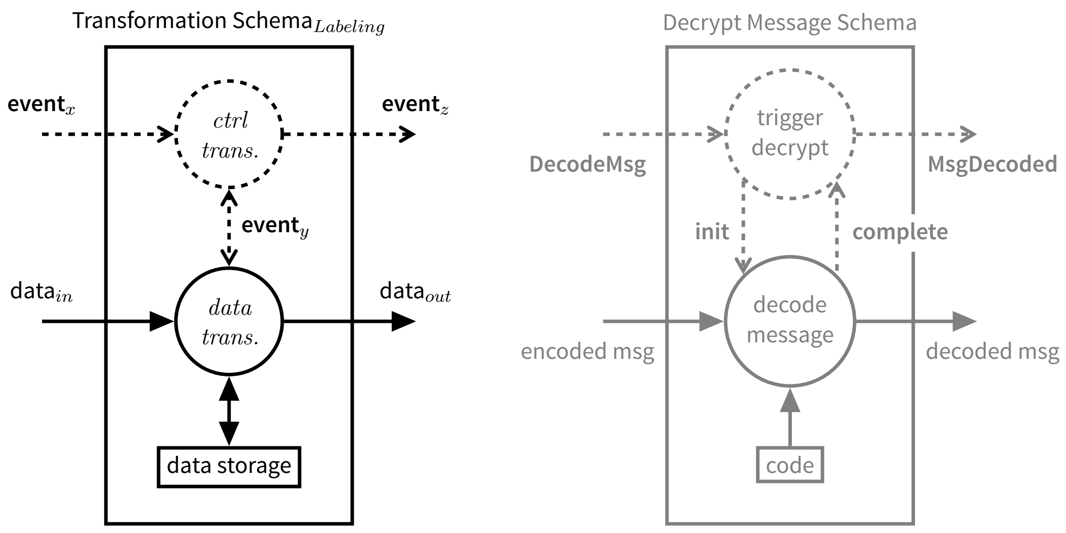


FIGURE 7.1 Transformation Schema, decryption example adapted from [217, p. 41, fig. 6.1]

The right, gray-colored side in figure 7.1 gives an examples for modeling the decoding of a message by means of a transformation schema. The data transformation *decode message* is taken from an example given by Ward and Mellor [217, page 41, figure 6.1] and put into a transformation schema named *Decrypt Message Schema*, which is executed as follows:

1. On appearance of event *DecodeMsg* the control transformation *trigger decrypt* starts its event processing. It creates event *init*, which activates the data transformation *decode message*.
2. This data transformation requires the *encoded msg* as input data, as well as the code used for decoding a message, which is available to the data transformation from the data storage *code*.
3. After the data transformation has decrypted the encoded message by use of the code, it provides the *decoded msg* as output data to other processes, which are described by respective transformation schemas.
4. Then, the data transformation *decode message* informs the control transformation *trigger decrypt* via event *complete* that processing of the encoded message is finished and desired output data is made available.
5. The control transformation *trigger decrypt* creates hereupon a notification event *MsgDecoded*, which indicates the availability of the decoded message.

At first glance, it may feel strange to care about an apparently oldfashioned modeling approach to early software development artifacts, one which obviously struggles with the same complexity issues as nowadays requirements analysis still does, such as caused by undefined granularity, combinatorial explosion, ambiguity due to missed commonalities, etc. as motivated in Part I Software Projects – Perspectives on a Managed Engineering Discipline and challenged in part Part II Problem-Based Project Estimating by introducing functional size measurement patterns and requirements work packages.

Combining transformation schemas with patterns for software analysis and software design helps to overcome the burden of uncontrolled complexity. This approach has not been tried out in literature and research so far as to my knowledge.

Subsequent sections prepare Transition templates, which represent the result of such a combination of schemas and patterns. These templates provide a means for coping with complexity, dependency, and variability related to functional software requirements throughout the software development project.

7.3. Transition Templates – Making problems absorb into platform

Transition templates form a new kind of pattern, which slices the twin peaks model [162] of intertwined problem analysis and solution design at a defined level of detail. They represent "patterns of patterns" that pave the way for creating any architectural blueprint for a given requirements work package.

In order to establish this capability of transition templates, the essential-model/implementation-model approach from Ward and Mellor [217, page 30, figure 4.1] – the precursor of the twin peaks model – is developed further.

For this purpose, the *processor* and *task* modeling, which belongs to the implementation-model (solution peak), and which serves the identification of "the processors that will carry out the [transformation] work" [219, page 19], are weaved into the concept of Ward and Mellor's Transformation Schemas as briefly introduced in section 7.2.2. Transformation schemas represent the "features" [219, page 20] or "a portion of the work [to be] done" [219, page 19] in the essential-model (problem peak).

That way tailored transformation schemas build transition templates, which guide "the decision-making processes that lead to allocation" [219, page 24] of desired software functionality to a software platform to-be. They help in "reorganizing the content of the essential model[, which is relatable to a functional requirements specification] to reflect the choice of a [candidate] processor configuration[, such as given by a blueprint of a software architecture]." [219, page 36].

Commonly known patterns for problem analysis and solution design are generalizable to this newly formed concept of transition templates. In doing so, transition templates provide for bridging the problem-solution peaks, which is illustrated by means of problem-based functional size measurement patterns as well as a set of pre-selected architectural design patterns in the following.

Table 7.1 establishes the context for a seamless handover, i.e. an allocation process, of problems identified in software analysis to candidate solutions developed in software design.

		problem peak	– **bridge** –	**solution** peak		
	Requirements Work Package	essential model	**transition template**	implementation model	Architectural Blueprint	styles & design **patterns**
FSM **patterns**	machine	one processor	processor	multiple processors	configuration (roles)	
	problem processes	multiple tasks	**task**	one task	solution candidate (type)	
	requirements		transformations		computational components	
	problem template, detailed in section 7.4		**"basic unit of activities"**	solution template, detailed in section 7.5		

TABLE 7.1 Bridging the problem-solution gap by transition templates

At the core of this approach and in the center of table 7.1 are transition templates, which build on the definitions of *processor*, *task*, and *transformation* as introduced by Ward and Mellor [217], and which merge into the idea of a "basic unit of activities", which is present in problem analysis as well as in solution design, and recognizable in connection with the use of patterns.

Section 7.4 designs transition templates for problems, such that the resulting problem templates represent "basic units of activities", which become tangible within requirements work packages. It details the first and second column of table 7.1.

Section 7.5 designs transition templates for solutions, such that solution templates represent those"basic units of activities", which are evident to architectural blueprints, and such that these units become identifiable. It provides more insights to the fourth and fifth column of table 7.1.

On these grounds, transition templates maintain a strong relationship between patterns in problem analysis and solution design, and thereby facilitate the allocation of desired software functionality as specified by the requirements to the computational components that configure the design of a candidate software architecture.

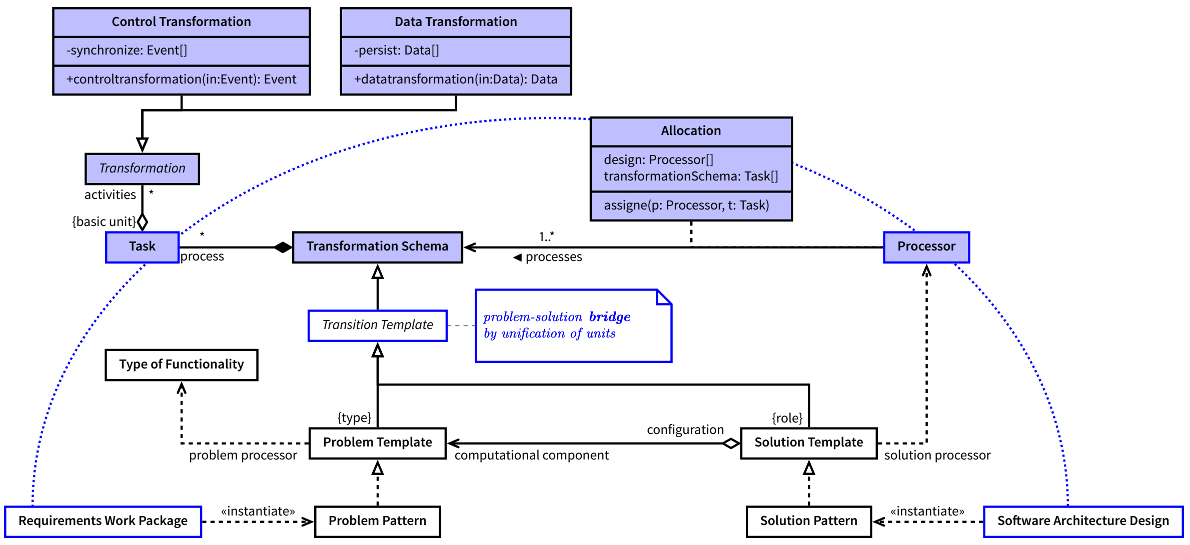


FIGURE 7.2 Transformation Schemas and their adaptation to Transition Templates

Figure 7.2 represents the use of concepts in this section by means of a UML class diagram, which starts in its left upper corner by the description of what is a *transformation*.

As introduced in section 7.2.2, according to Ward and Mellor's understanding, a transformation is classifiable into either a *control transformation*, which describes the processing of events, or a *data transformation*, which describes the processing of data.

These are combined within a *transformation schema* for expressing some desired software functionality. The kind and number of transformations combined to a transformation schema is at will of a modeler. This gives reasons for using an asterisk * as multiplicity in this aggregation relation of transformations and transformation schema.

Ward and Mellor define a *task* as a "set of instructions that is manipulated [. . .] as a unit by the [. . .] processor" [219, page 6]. It represents "any named, independently scheduable piece of software that implements some portion of the transformation work assigned to a processor. " [219, page 37]. This set of instructions or portion of transformation work is put on a level with the transformations that are combined to transformation schemata. Respectively, transformations take the role of activities that are united by a task. That is the reason for making the class *task* an intermediary one between the classes *transformation* and *transformation schema*. It represents an atomic or "basic unit of activity" [219, page 37] or *process*, which builds on transformations.

It is the definition of "basic unit", which is challenged in this dissertation. This pre-defined unit of software functionality must be comparably meaningful in software analysis and software design for bridging the problem-solution peaks, respectively. This provides for establishing an *allocation* process, that assignes each task to a proper processor, which is "a person or a machine that can carry out instructions and store data" [219, page 6]. In the following, this allocation process is implemented by the use of patterns for mapping a problem process as given by desired software functionality (requirements specification) to a proper solution processor out of the computational components available in the (architectural design of a) software to-be.

The lower parts of figure 7.2 are detailed by section 7.4 Problem templates and section 7.5 Solution templates.

7.4. Problem templates

Figure 7.2 Transformation Schemas and their adaptation to Transition Templates on page 118 shows in its lower left, that a *problem template* as developed next is a pattern, which exhibits selected properties of a *transformation schema*. That way, problem templates structure software functionality into a recognizable problem *process* or *task* respectively, which relates to a defined *type of functionality*.

To this extent, problem templates conform to *problem patterns*, such as problem-based functional size measurement patterns. Both apply to software analysis for the grouping of desired software functionality as is available in a requirements work package.

In contrast to problem patterns, *problem templates* add further constraints to the grouping of interactions, i.e. the different tasks involved with a problem at hand. As is elaborated in chapter 8 Problem-Based Adaptation Framework, this makes problem processes and their related requirements dependencies better tangible.

As shown by figure 7.3, problem templates limit the number and ordering of interactions, which is achieved by restricting the modeling of tasks as representable by a transformation schema to particular uses of notational elements. A problem template consists of exactly one control transformation (dashed circle) and one data transformation (solid circle).

The control transformation accepts exactly one *trigger*ing input event, and produces at maximum one *action* event, which is made available to other tasks. The data transformation reads input *data* and can produce data as a *result*, which is made available to other tasks. It can also store and access data from a storage, which exhibits the *state* of the task. Control and data transformations of one problem template send events among one another. These events internal to a task must be relatable to its input trigger and output action event.

In contrast to a transformation schema, a problem template does not make use of storages for events. In this work, the synchronization of events is shifted to the consideration and means used for expressing the software life cycle. It thus happens at a higher level than that of one task. For establishing the software life cycle, events are synchronized for modeling a (temporal) ordering relation among tasks. This topic belongs to chapter 8 Problem-Based Adaptation Framework.

As a result, problem templates ensure a grouping of software functionality and therewith related requirements into meaningful, and independent **tasks** on a pattern basis. They represent the smallest possible unit of transformation work, which is in conformance with Ward and Mellor.

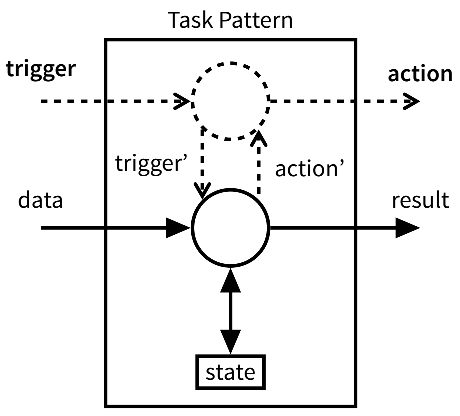


FIGURE 7.3 Problem template structure

7.4.1. Set-Up Problem Templates

Problem patterns help to classify some software functionality, such as involved with a requirements work package (RWP), at the level of recognizable problems by taking primarily their static properties[3] into account. At the heart of each problem is the machine, which takes the role of a problem processor that is capable of operating a particular type of software functionality.

Problem templates represent *task patterns*, which likewise help to structure the requirements that belong to a known problem further. These patterns focus on the dynamic properties[4] (flow logic) involved with some software functionality (task), and constrain its sequence of actions. Each forms a usage *scenario* [14] at the level of recognizable problem processes, which specifies a "basic course of action" [12, page 321] of how to satisfy desired software functionality.

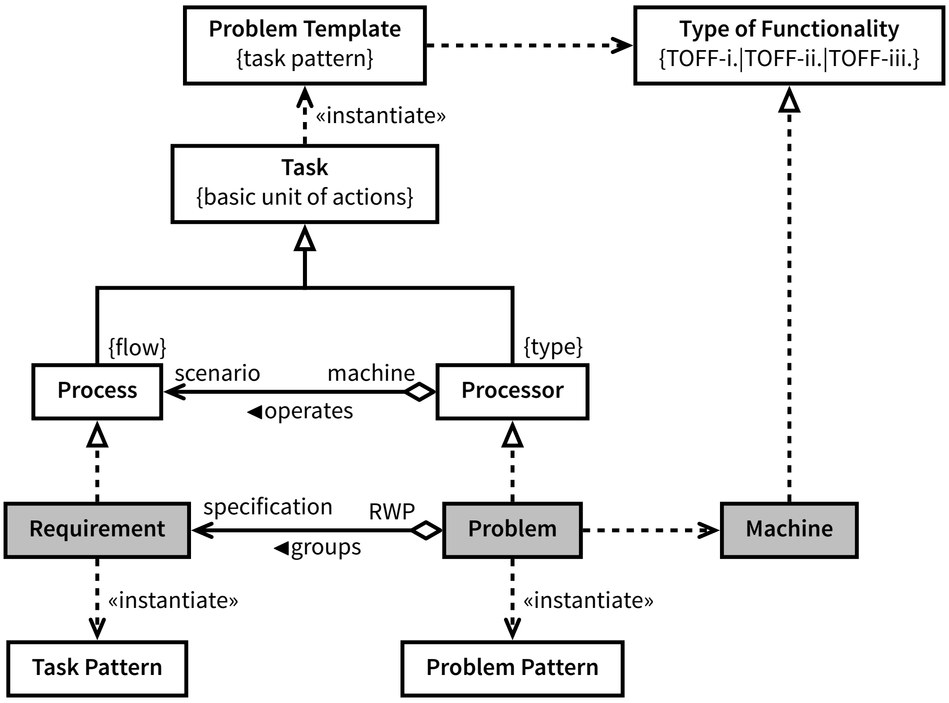


FIGURE 7.4 Problem templates are task patterns

Consequently and as figure 7.4 shows, problem templates bring the concepts of structured analysis together with those of problem-oriented software engineering. Task patterns map Ward and Mellor's understanding of a *processor*, which is in the position to operate *processes*, and their shared relation to and meaning in the context of *tasks* to the concepts as defined by Jackson. The meaning of the *machine*[5] for deriving *specifications*, and for classifying and thus for establishing a(n independent) grouping of *requirements* is thereby further analysed, i.e. how satisfiable, desired software functionality is representable by basic units of machine interface specifications is defined.

In the following a three-step procedure is described and executed, which explains the methods and models used for establishing problem templates.

[3] *The what?* regarding the kind and number of elements that belong to one problem

[4] *The how?* regarding the relationships of elements, which belong to one problem, i.e. their temporal dependencies

[5] The machine interface of a *problem* outlines a plan (specification) for satisfying the requirements; it exhibits a particular type of functionality

About problems, and their type and flow inside of involved processes

Problem templates are task patterns, which characterize an independent, problem-based unit of activities, whose execution serves to fulfill a defined set of requirements. Each of these pattern restricts the flow of activities involved with the problem unit, i.e. its tasks, in regard to the type of functionality, which is specific to the problem at hand and already designed in section 5.4.3.

The use of these task patterns is twofold: first, they set up recognizable units of activities, which are meaningful for constituting problems as well as solutions. That way, they provide for bridging the gap between these different perspectives in software development works, as is illustrated in section 7.5 Solution templates of this chapter.

Second, they set up independent units of activities, which are meaningful for modeling equally formed usage scenarios. These represent pre-defined units for deriving uniform requirements specifications, which help to keep the level of detail for expressing these consistent.

Considering problems and consequently their covered requirements at the task level, eases the sequencing of related software functionality. Respectively, modeling their dependencies and deciding on their composition as is done and documented by the software life cycle, which is topic to chapter 8, becomes much more manageable.

According to Mapping Patterns to Processes by Types of Functionality on page 52, there are three types of functionality (TOFF-i. to -iii.), which are characteristic to problems and their involved requirements that are produced by means of problem frames or problem-based functional size measurement patterns.

Section 7.4.2 presents the Problem Template for TOFF-i problem processes. It elaborates the task pattern, which underlies problems, whose primary concern is to process information, which is received. Section 7.4.3 presents the Problem Template for TOFF-ii problem processes. It elaborates the task pattern for problems, whose primary concern is to provide information, which is retrieved. Section 7.4.4 presents the Problem Template for TOFF-iii problem processes. It elaborates the task pattern for modeling problem processes, whose primary concern is to provide information, which is derived.

These three problem templates are developed by the following procedure and models:

1. **Account for the problem.**

   The general structure of (Jackson's frame diagrams that underlies) a requirements work package is of first interest. This is why each discussion of a problem template involves a respective diagram, such as given by figure 7.6 TOFF-i. problem (general structure) on page 127, which identifies relevant domains and their interactions.

   This structure is responsible for mapping a set of requirements into a (static) problem description, one which fits a known type of functionality[6], e.g. TOFF-i., TOFF-ii., or TOFF-iii.

   In this context, the machine domain is of special interest. It takes the role of a problem processor, which is in the position to operate units of activities (problem processes) as defined at the machine's interfaces, and that possess a particular type of functionality.

   Thus, the machine processes an ordered sets of desired software functionality (tasks) respectively to satisfy the requirements. "The machine is what must eventually be built and installed to solve the problem." [128, page 15]

[6]a problem's type of functionality is apparent by use of problem(-based functional size measurement) patterns

2. **Account for the type of a problem process.**

   The type of functionality that characterizes a problem determines the structure of a problem template. For instance, the primary intent of a TOFF-i. problem is to store some received data, cf. figure 7.7 TOFF-i. processor (task pattern) – Template for operating TOFF-i. processes on page 127. Accordant notational elements of a transition template as designed in section 7.3, which represent a customization of Ward and Mellor's transformation schemata as introduced in section 7.2.2, are chosen to set up a respective problem template. It models this problem concern as a task pattern.

   A problem template describes the standard way (specification) of how a machine (problem processor) executes a task (problem process), that belongs to a particular type of functionality (problem).

   It defines a unification for those units of computation, which are inherent to patterns of software analysis as well as those of software design. That way, the structuring of problems into processes by problem templates is of twofold use:
   First, it enables the identification of corresponding processors in any architectural blueprint and thus the exploration of solution alternatives, and it also guides the mapping of desired software functionality to the computational components of a candidate software (architecture) design. This advantage of problem templates is detailed further in section 7.5 Solution templates.
   Second, considering problems at the level of their processes or tasks enables better control of involved dependencies. This is of great importance for managing the software life cycle, which provides the basis for the decision making on project plans, and which is elaborated in depth in chapter 8 Problem-Based Adaptation Framework.

3. **Account for the flow inside of a problem process**

   In addition to each problem template, there is a task scenario available, cf. figure 7.8 TOFF-i. process (general structure) on page 127. It models the temporal order of transformation work that is covered by a task pattern using OMG's UML sequence diagrams.

   Each problem process binds domains and requirements of a problem unit, which are involved with a task, to one alt-fragment. Each of these defines a sequence of how the machine (problem processor) operates the different activities of a task (problem process) to satisfy the (problem unit of) requirements.

   Alt-fragements are used in here for representing a basic unit of activities that make one independent and complete task. To emphasize this fact, only synchronous communication is used at the machine life line. Chapter 8 Problem-Based Adaptation Framework elaborates the method and benefits from this practice for establishing the project plan. Knowing the flow inside of a problem process helps to create uniform task descriptions and to derive respective specifications consistently.

Mapping problems, tasks, and scenarios

Table 7.2 represents a mapping of conceptions from problem-oriented software engineering based on Jackson's problem frames approach [128] to those models used by Ward and Mellor in structured analysis (transformation schemata) [218], as well as OMG [168]'s tools (sequence diagrams) for object-oriented software analysis [168].

This mapping allows for a consistent transformation of models between these three disciplines, which establishes traceability between requirements and specifications. On the one hand, table 7.2 summarizes the findings of subsequent sections, and thus anticipates important contributions as actually developed next. On the other hand, it provides information, whose application is of more general use, than only for producing problem templates. For instance, it can be used for modeling tasks and scenarios based on any known problem frame. It is not limited to the use of problem-based functional size measurement patterns, or applicable only for building task descriptions that reflect one specific type of software functionality.

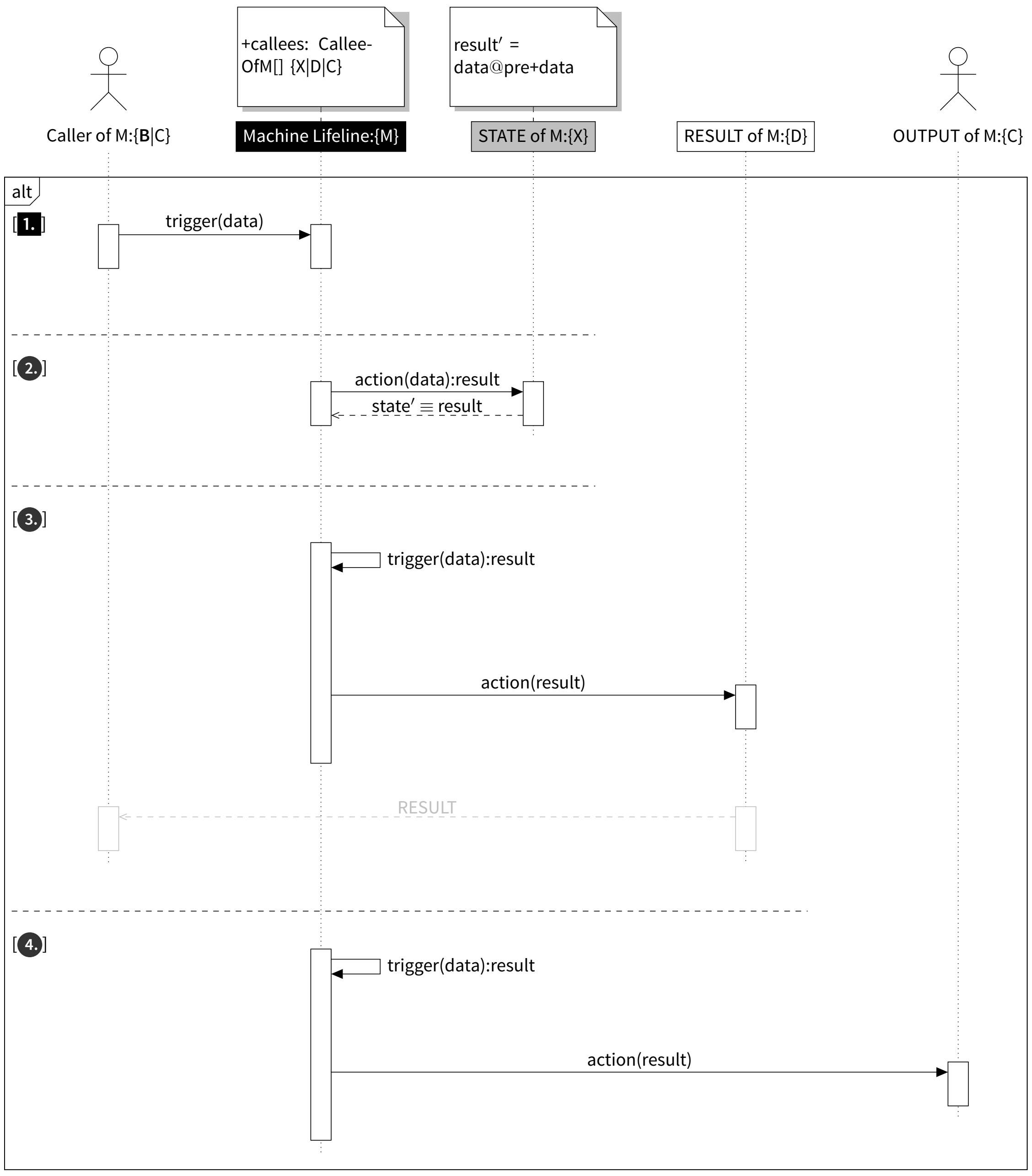
machine interfaces with type of problem domain	**machine** ***observes*** type of phenomenon	machine interface as given by problem templates, cf. figure 7.4	**machine** lifeline in UML sequence diagram ***receives*** message	details cf. fig. 7.5
biddable domain (B)	causal	input event *trigger*	call msg::= trigger(data)	❶
	symbolic	input data		
causal domain (C)	causal	input event *trigger*	call msg::= trigger(data)	❶
	symbolic	input data		
display domain (D)	causal	*n/a*	*n/a*	
	symbolic	*n/a*		
lexical domain (X)	causal	*n/a*	reply msg::= state’ = *action(data):result*	❷
	symbolic	state’ *storage box*		

machine interfaces with type of problem domain	**machine** ***controls*** type of phenomenon	machine interface as given by problem templates, cf. figure 7.4	**machine** lifeline in UML sequence diagram ***sends*** message	details cf. fig. 7.5
biddable domain (B)	causal	*n/a*	*via connection domain display (D)*	
	symbolic	*n/a*		
causal domain (C)	causal	output event *action*	call msg::= action(result) = *trigger(data):action(result)*	❹
	symbolic	output result		
display domain (D)	causal	action event	reply msg::= result = action(data):result	❸
	symbolic	output result		
lexical domain (X)	causal	action event	call msg::= action(data):result	❷
	symbolic	input data		

TABLE 7.2 Basic activities at the machine interface

Table 7.2 presents four basic activities that occur at the machine interface, for specifying satisfiable, desired software functionality. Utilizing these results in reproducible requirements specifications, which are modeled in a transformation schema style, or by means of UML sequence diagrams.

Figure 7.5 details the four basic activities **1.** to **4.** by representing these as UML sequence diagram fragments.



Legend: B=biddable, C=causal, X=lexical, D=display problem domain, M=machine domain

FIGURE 7.5 Model kit of basic activities for creating specifications by UML sequence diagrams

The following sections elaborate combinations of these basic activities to form problem templates. These are patterns applicable for modeling tasks (basic units of activities) and respective scenarios (flow of activities within a basic unit) for those problem descriptions, which involve a defined type of software functionality (TOFF-i. to TOFF-iii.).

7.4.2. Problem Template for TOFF-i.

The problem illustrated by figure 7.6 is concerned with software functionality that is classifiable into TOFF-i., which is in general concerned with the processing of some received information. Section 5.4 Problem Class – Kind of Functionality elaborates and discusses in detail what this kind of problem classification is about. There are some constraints on the setup of a respective problem template (task pattern) for a TOFF-i. problem, which must be taken into account:

- The problem template for TOFF-i. tasks conforms to an elementary process known from function point analysis, whose application boundary comprises only those software functionality, which represents an external input (EI).

  That is, the task pattern that is to be built, must combine transformations in a way, such that it forms a unit of basic activities, which describes a machine (problem processor) that is capable of operating TOFF-i. software functionality, i.e. a problem process that conforms to an EI.

- The main purpose of an external input (EI) is to processes information, which is sent from outside the application boundary to the elementary process, and stored to an internal logical file (ILF). The processing of information and its results remain inside the application boundary.

  This fact is taken into account by associating the lexical problem domain given in the problem model with the data storage box in the task model. Inside the TOFF-i. problem template this data storage box models the ILF. It can only be accessed by the data transformation, which belongs to the TOFF-i. machine. That means, any change to an ILF, and respectively to information, which are advantageous and used in the following for representing the state of a software application (life cycle), is under exclusive control of this problem processor.

- Other problems or types of elementary processes can inquire an internal logical file (ILF).

  Yes, but they cannot change it, when making use of task patterns[7] as designed here. Inquiries to an ILF or respective reads of a lexical problem domain demand another elementary process and characterize a different type of functionality, namely those of TOFF.-ii. and TOFF-iii., which are discussed in upcoming paragraphs.

Figure 7.7 presents a problem template (task pattern) applicable to (and thus operatable by) TOFF-i. problem (processor)s. Accordingly, the machine and its interfaces as given in the problem description that is outlined in figure 7.6 are expressed by means of a customized transformation schema as introduced in section 7.2.2 for assembling the task pattern in figure 7.7. It comprises the following activities:

1. A $trigger$-event with respective $data$ is received by the problem processor. That is, this input event and respective input data enter/are in the scope of responsibility of a TOFF-i. machine (black-filled box).
2. The control transformation (dashed circle) processes the $trigger$ and initiates an $action$-event, which in this problem template controls the data transformation.
3. Depending on the $action$-event, the data transformation (solid circle) within this problem template starts processing the $data$ received, which produces a $result$.

[7] Note: Problem templates are designed to condense problems as given by requirements into tasks, which are basic units of activities (or transformation work).

4. The data transformation makes the *result* of this data processing persistent in a data storage named *state*. That means, the *result* resides inside the scope of responsibility of this TOFF-i. problem processor. The *state* box is a kind of internal logical file (ILF) that belongs to this problem, i.e. it represents the respective lexical domain.
5. The *state* box represents the persistent data that can be written and read by a data transformation. Depending on what information is stored to this *state*, i.e. by means of the available result data, it becomes possible to synchronize different tasks, and thus to set up a software life cycle.
6. This TOFF-i. problem (processor) in itself does not provide any triggering-events or resulting data to other tasks. As already mentioned above as constraint on a TOFF-i. problem template, the processing of information and its results (in regard to the writes of an ILF) remains inside the scope of responsibility of this task.

The scenario in figure 7.8 arranges the four basic activities as summarized by the combined fragments in figure 7.5 into a flow of activities, which is bounded to the execution of a specific task, namely to those activities, which are of relevance to satisfy (the requirements of) TOFF-i. problems. That is, each task scenario relates to requirements as defined in the respective problem description. This relation is revealed by use of a UML note symbol at the respective fragement. Usually, there are several instances of task patterns and respective scenarios to cover the entire, desired software functionality that is involved with a TOFF-i. problem description. At the heart of each scenario is again the machine (lifeline of the problem processor), which establishes the flow logic of activities as specified at its interfaces. The scenario in figure 7.8 comprises the following activities, which result from instantiating the TOFF.-i. problem template given in figure 7.7 in the context of a TOFF-i. problem description as in figure 7.6:

1. The *machine* object represents the problem processor (marked filled black), which is in the position to execute software functionality that is characteristic to TOFF-i. problems. It receives (from the *activator object* of the environment) a synchronous *trigger*-message with some *data* as argument.
2. The *machine object* processes the call and data (via its control transformation).
3. This (event) processing of the *trigger*-message (white-filled machine life line) involves the sending of a synchronous *action*-message together with the *data* to the machine's *state object*.
4. The *state object* is in charge of (the data transformation, gray-filled life line) producing the desired *result* and to make it persistent.
5. The state object represents the *state* of the problem, which is known in first instance to the machine object aka the problem processor only.
6. This use scenario of a TOFF-i. problem process does not demand any (reply) messages from the machine object to its calling activator object (in the environment). Nevertheless, other types of problem processes can access state information.

Section 7.4.1 summarizes the findings gathered in here for Mapping problems, tasks, and scenarios. Problem templates are used in section 7.5 Solution templates for mapping patterns of problem analysis with those of solution design, and in chapter 8 to form units of activities, which are beneficial for expressing the software life cycle. This is hardly achievable on the basis of UML sequence diagrams only, if these are not bound to comparable units of interaction. Task scenarios are used to produce uniform requirements specifications, which are based on patterns, and the preliminary work on requirements estimating by functional size measurement in part II. They define interactions that share the same level of granularity and purpose, and the participating objects, e.g. other users or tasks, involved with these.

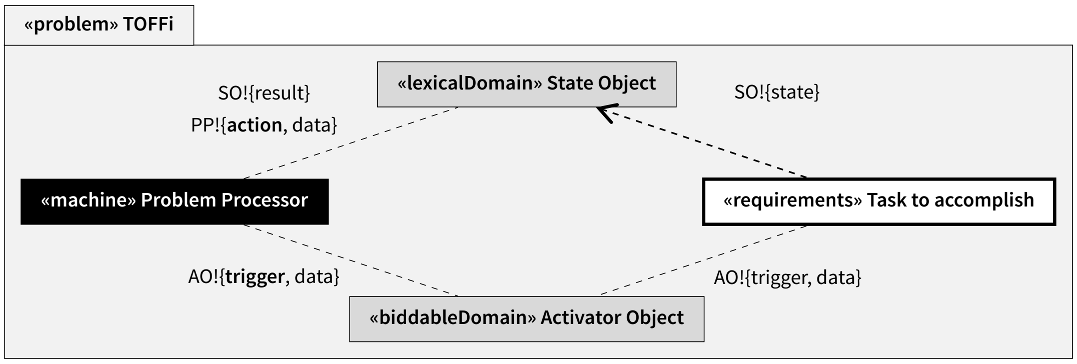

«problem» TOFFi
«lexicalDomain» State Object
SO!{result}
PP!{**action**, data}
SO!{state}
«machine» Problem Processor
«requirements» Task to accomplish
AO!{**trigger**, data}
AO!{trigger, data}
«biddableDomain» Activator Object

FIGURE 7.6 TOFF-i. problem (general structure)

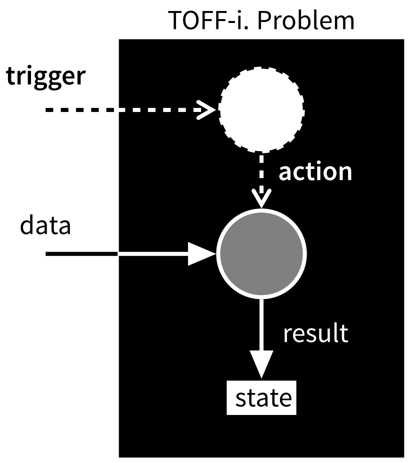


FIGURE 7.7 TOFF-i. processor (task pattern) – Template for operating TOFF-i. processes

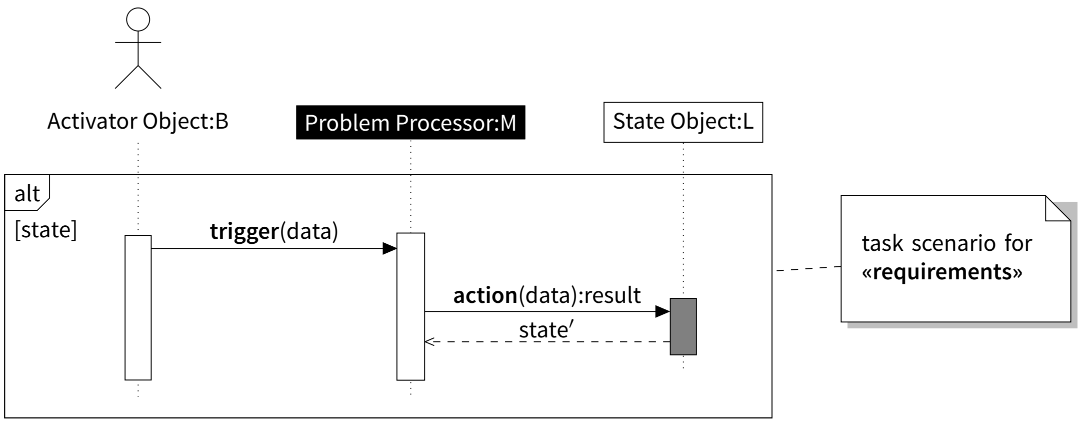


FIGURE 7.8 TOFF-i. process (general structure)

7.4.3. Problem Template for TOFF-ii.

The problem modeled by figure 7.9 involves software functionality, which is of a TOFF-ii. type. It is in general concerned with providing some information to the environment (via a display domain) that is retrieved by the machine, see definition **DEFINITION 5.3** on page 52.

There are some constraints on the setup of a respective problem template (task pattern) for a TOFF-ii. problem, which must be taken into account:

- The problem template for TOFF-ii. tasks conforms to an external inquiry (EQ) elementary process as known from function point analysis.

  The task pattern to be built must combine transformations in a way, such that the respective machine (problem processor) is capable of operating this kind of software functionality.
- The main purpose of an EQ is to retrieve information and simply showing these. Therefore, it can access any data storages available, either internal to the application under consideration (ILF), or external by means of interfaces to participating objects or other tasks (EIF).

  In contrast to the task pattern for TOFF-i. problems as discussed before, the primary purpose of TOFF-ii. problems is not to change data. Here, the focus of data processing is to assemble given information for providing some new knowledge only.

The problem template in figure 7.10 represents a task pattern for operating TOFF-ii. problem processes. It expresses the interactions at the machine interface taken from the problem description in figure 7.9 by a customized transformation schema as follows:

1. A $trigger$-event with respective $data$ is received by the problem processor (machine, black-filled box).
2. The control transformation (dashed, white-filled circle) of the machine processes the $trigger$. It initiates a respective $action$-event to control the data transformation (solid, gray-filled circle).
3. Depending on the $action$-event, the data transformation starts processing the $data$ received[8], which produces a $result$.
4. The data transformation makes the $result$ of its data processing available. It can be said, that the result leaves the scope of responsibility of this task and is made available for further processing. To this end, the TOFF-ii. problem processor in itself remains stateless after the data processing. This TOFF-ii. problem process does only provide data information. It is not concerned with producing control information.

[8]Note: The "data received" can be any information retrieved by the machine from other users, tasks, or respective data storage boxes.

The UML sequence diagram in figure 7.11 shows a scenario or specification of the flow logic for activities at the machine interface, which gives an interpretation [9] of the TOFF-ii. problem template in figure 7.10 in the context of a TOFF-ii. problem description such as present in figure 7.9:

1. The $machine$ object represents the problem processor (marked filled black), which is in the position to execute software functionality that characterizes TOFF-ii. problems. It receives (from the $activator\ object$ of the environment) a synchronuous $trigger$-message with some $data$ as argument.
2. The $machine\ object$ processes the call and data (via its control transformation). This (event) processing of the $trigger$-message (white-filled machine life line) invokes an $action$-self message together with the $data$ received from the activator object.
3. This self message is a placeholder for the activities to be done by the machine (i.e. relevant data transformations, gray-filled machine life line) for retrieving and composing those data information that make the result.
4. The $output\ object$ (display domain) is in charge of presenting the calculated $result$ to the environment, e.g. to that activator object, which has triggered this task. That is why there is in light-gray color a return message between the display and the biddable domain. Since it cannot be controlled by the machine, that the biddable domain makes use of the $result$, this reply message should only indicate that the result is allotted to the problem domain, which calls the machine. The display domain takes the role of a connection domain between the machine and the activator object.

The $[state]$-condition used for the alt-fragement represents a precondition for accomplishing this task. It refers to a $state$ as can only be established by TOFF-i. problems, see section 7.4.2 on page 125. Nevertheless, this TOFF-ii. problem process is stateless, it does not affect the $state$ in itself. Mentioning the $[state]$-condition here is of use only for synchronizing different tasks, which ultimately makes the software life cycle. Details on this topic are elaborated and discussed in respective sections.

[9] Note: Realistic TOFF-ii. problems involve several problem domains from which data is retrieved. This fact is not denied, and as shown by the examples in the Case Studies part, these problem domains are incorporated in the scenarios, too. In the example given here, the question to be answered is what software functionality makes the core of TOFF-ii. problems and respective tasks.

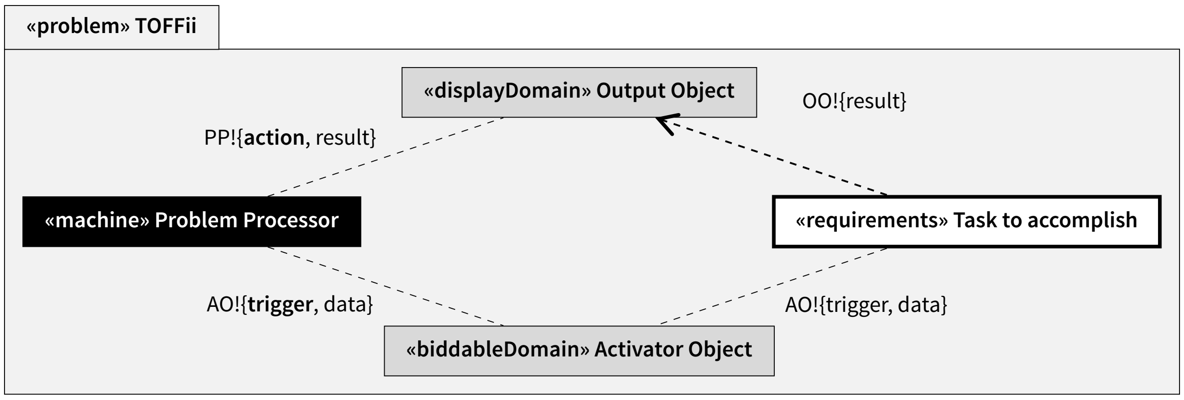

«problem» TOFFii
«displayDomain» Output Object
OO!{result}
PP!{action, result}
«machine» Problem Processor
«requirements» Task to accomplish
AO!{trigger, data}
AO!{trigger, data}
«biddableDomain» Activator Object

FIGURE 7.9 TOFF-ii. problem (general structure)

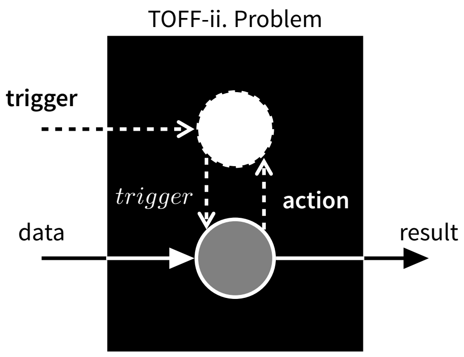


FIGURE 7.10 TOFF-ii. processor (task pattern) – Template for operating TOFF-ii. processes

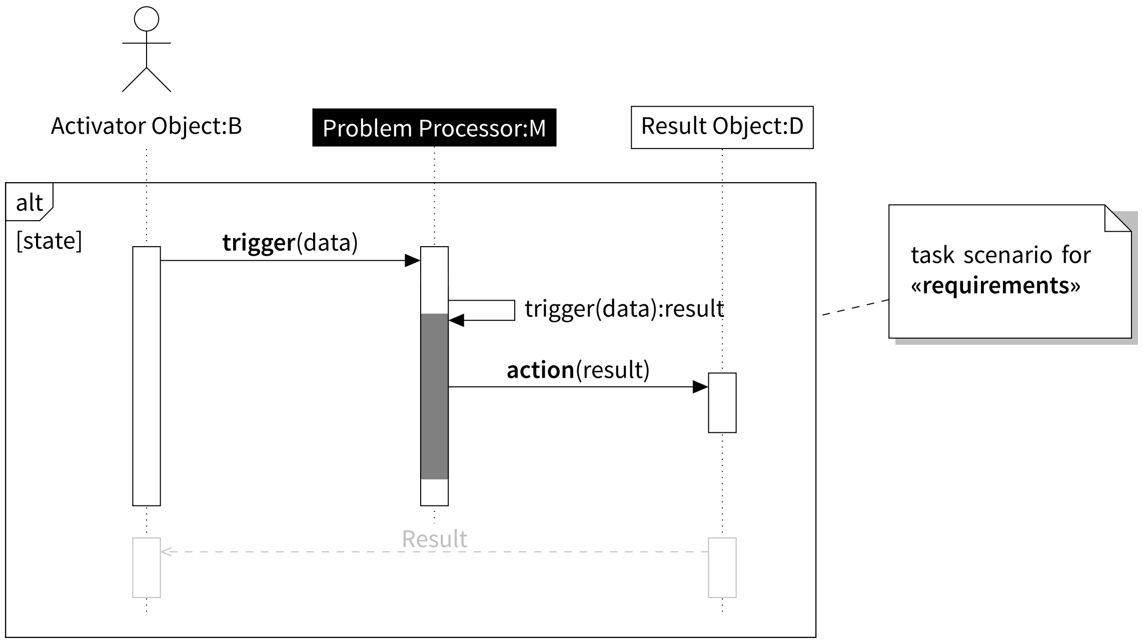


FIGURE 7.11 TOFF-ii. process (general structure)

7.4.4. Problem Template for TOFF-iii.

The problem modeled in figure 7.12 is characteristic for software functionality, which fits TOFF-iii. In general, its primary purpose is to present some information (to the environment), which is calculated or derived by the machine, cf. definition **DEFINITION 5.3** on page 52. In contrast to TOFF-ii. problems, a TOFF-iii. problem involves a higher effort of computation regarding the delivery of desired information, than simply retrieving these from somewhere.

The following constraints for setting up a TOFF-iii. problem template (task pattern) is taken into account:

- The problem template for TOFF-iii. tasks conforms to an external output (EO) elementary process as known from function point analysis.

  Figure 7.13 maintains the primary purpose of an EO, which is characterized as having a higher computational effort than an external inquiry (EI) process, i.e. a TOFF-ii. problem. This fact is reflected by the data transformation, which takes over the role and respective responsibilities of the control transformation, and thus is concerned with control information in addition to the processing of data information. Furthermore, the initiation of the $action$-event by the data transformation can be regarded as indication (signal) for the completion of the computational task. This is another detail, which distinguishes TOFF-ii. and TOFF-iii. task patterns, since the TOFF-ii. problem template and its respective machine delivers only data information as reply to a call. The TOFF-iii. problem template delivers control information and data information as reply to a call.

The problem template in figure 7.13 represents a pattern for modeling tasks, which can be operated by a TOFF-iii. problem processor:

1. A $trigger$-event with respective $data$ is received by the machine (black-filled box).
2. The control transformation (dashed circle) as part of a TOFF-iii. task acts as dummy. It simply passes the $trigger$ over to the data transformation, but does not process it any further.
3. It is the data transformation (solid, gray-filled circle), which deals with the processing of the $trigger$-event, that it has gotten from the control transformation. Depending on the received $trigger$, it determines the respective $action$ that is to initiate and processes the $data$ received for the calculation of the $result$.
4. Due to its responsibility for the $result$ as well as the $action$-event, the data transformation can be considered as to have an added computational effort, namely those of processing control and data information. After completing its computation, the $result$ and belonging $action$-event leave (via the dummy control transformation) the scope of responsibility of this problem processor.

The $action$-event provides context to the $result$. It serves together with the respective result data as trigger to the environment. A TOFF-iii. problem processor provides control and involved, derived information to other tasks.

The UML sequence diagram in figure 7.14 specifies the flow of activities, which are characteristic for a TOFF-iii. problem. It is an interpretation of the TOFF-iii. problem template in figure 7.13 in the context of a TOFF-iii. problem description such as given in figure 7.12:

1. The *machine* object represents the problem processor (marked filled black), which is in the position to execute software functionality that characterizes TOFF-iii. problems. It receives from the *activator object* of the environment a synchronuous *trigger*-message with some *data* as argument.
2. The *machine object* processes the call and data (via its control transformation). This (event) processing of the *trigger*-message (white-filled machine life line) invokes an *trigger(data):result*-self message, which hands over the processing of control information to the data transformation (gray-filled machine life line part).
3. This self message is a placeholder for all the activities to be done by the machine, i.e. relevant control and data transformations, for deriving data information that make the *result*, as well as control information that form an *action*-event.
4. The resulting output event and data form a reply message *action(result)*, which is sent from the *machine* in return to the call of the *activator object* and which is received by a *domain object* in the environment.

The interface and respective messages from the machine to the causal domain object can be used for simple transmission of data information or it can be understood as signal (control information), which performs some kind of control on the domain object. The second case makes the difference between TOFF-ii. and TOFF-iii. problem processes.

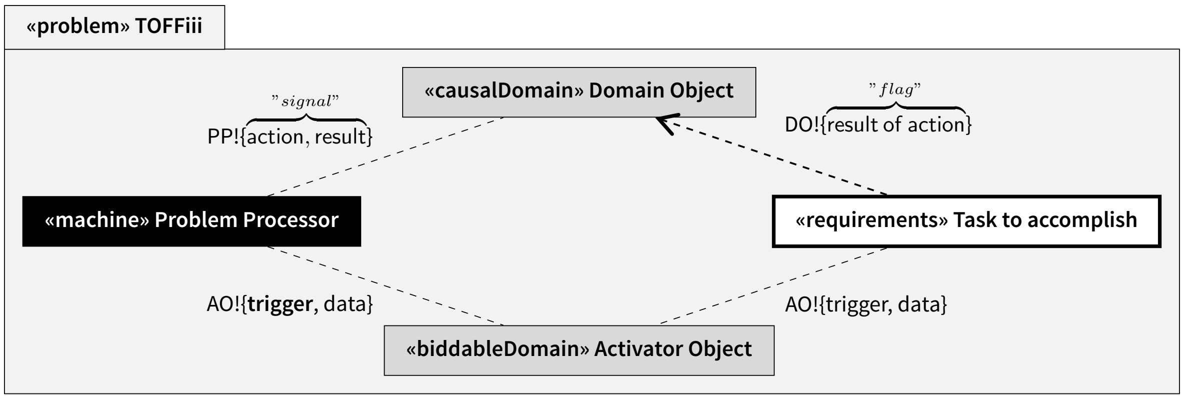

«problem» TOFFiii
«causalDomain» Domain Object
"signal"
PP!{action, result}
"flag"
DO!{result of action}
«machine» Problem Processor
«requirements» Task to accomplish
AO!{trigger, data}
AO!{trigger, data}
«biddableDomain» Activator Object

FIGURE 7.12 TOFF-iii. problem (general structure)

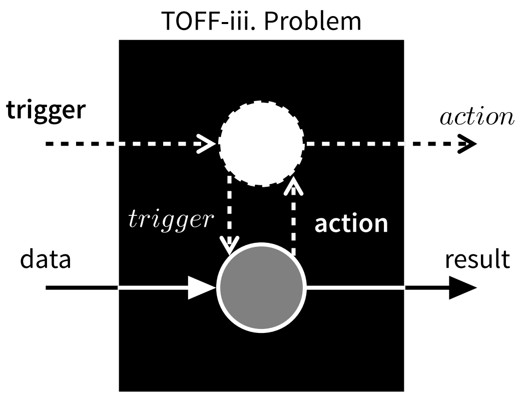


FIGURE 7.13 TOFF-iii. processor (task pattern) – Template for operating TOFF-iii. processes

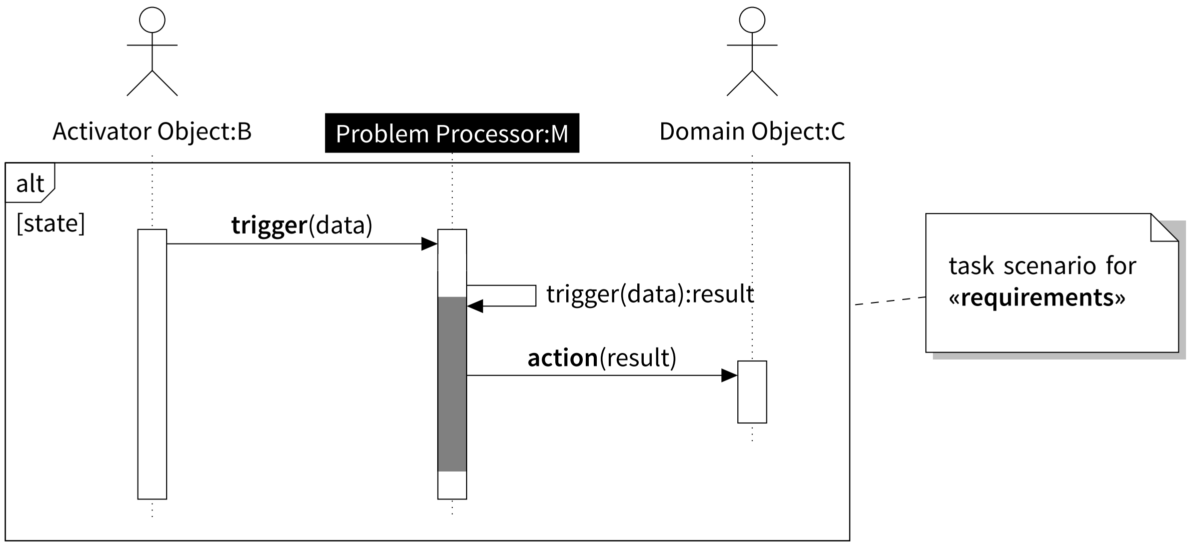


FIGURE 7.14 TOFF-iii. process (general structure)

7.5. Solution templates

Figure 7.2 Transformation Schemas and their adaptation to Transition Templates on page 118 shows in its lower right, that a *solution template* as developed next is a pattern, which describes the configuration of computational components that make a software's architecture design. It exhibits the *solution processors*, i.e. the interacting roles that share responsibility for the processing of a task.

That way, the *allocation* of desired software functionality to a software platform to-be becomes possible on a pattern basis, since the units of activities for describing problems as well as for describing solutions are comparably defined.

A solution template is a *configuration pattern* of patterns, which composes problem templates as introduced in the previous section to emulate best practices solutions, such as given by commonly known architectural styles and design patterns as listed in the appendix D Overview on Architecture Design Patterns.

7.5.1. Set-Up Solution Templates

The development of solution templates does not require the definition of a new notation, such as done for problem templates. Solution templates utilize problem templates to identify the tasks that can be processed by an architectural design. This is why problem templates and solution templates are both a kind of transition template, which is a specialization of Ward and Mellor's transformation schemata according to figure 7.2 in section 7.3.

The notational means for problem templates are reused to capture the solution, which is hidden in Architectural Design Patterns (ADP). An architectural structure can be seen as a composite of *computational components* that vary in their "classes of functionality and interaction they provide. [...] Software developers would clearly benefit from having more precise definitions of these [...]" according to Shaw and Garlan [198, Page 149].

Different classes of functionality have already been elaborated in detail from the domain of problem analysis. Three types of functionality, namely TOFF-i. to TOFF-iii. have been developed, and each problem template accounts for exactly one of these.

From the domain of solution design originates a classification developed by Shaw and Garlan [198], who suggest five component types[10] "that appear regulary in architectural descriptions" [198, page 149], where the following three are of further interest: Memory, Link, and (Pure) Computation. These are unique in their kind of processing data or signals, which conforms to the categorization according to TOFF as developed in this work.

- **Memory**: is a "shared collection of persistent structured data" [198, Page 149].
- **Computation**: is a "simple input/output relation [...with] no retained state" [198, Page 149].
- **Link**: "transmits information between entities" [198, Page 149].

The three respective rows in table 7.3 equate Shaw and Garlan's understanding of a *computational component* to that of Ward and Mellor's *processor* concept as used by this work. Thereby, it establishes a hierarchy into which ADP can be arranged, and which eases their differentiation into those which are architecturally significant, and those which are not.

[10]These classes of components are: (Pure) computation, Memory, Manager, Controller, Link. The two computational components *Manager* and *Controller* fit to Ward and Mellor's general concept of a *processor* and Jackson's *machine* domain. Both are universal in their kind of processing data or signals, and do not constitute a defined class of functionality.

Computational Component	Processor (black box)	Processor (white box)
Memory Computation Link	TOFF-i. TOFF-ii. TOFF-iii.	Data Storage Data Transformation Control Transformation
level of design details	coarse-grained	fine-grained
structures	architecture design by different types of processors	component design by different types of transformations
pattern classifiable as	architectural patterns and styles	design (and coding) patterns

TABLE 7.3 Patterns used for coarse-grained design are architecturally significant

In its second column, table 7.3 maps coarse-grained solution design to the consideration of interactions as provided by processors, which represents the level, which is of relevance to software architecture. In its third colummn, table 7.3 maps fine-grained solution design to the consideration of individual parts of a processor, which represents a level that is of relevance to particular components of a software architecture only.

The idea is to consider an ADP as a particular composition of processors, in which each of these computational components takes a specific role to accomplish a task. Within a solution design, several processors share the responsibility (and thus interact) for (the fulfillment of) a computational task. This coincides with Ward and Mellor's view on the shared processing of some transformation work, who suggest that "The name given to a processor should highlight the *role* played by the processor in the computation" [219, page 23]. This view on an architectural blueprint is complemented by table 7.1 on page 117.

The interactions provided by the computational components of an architecture design are first bound to their type of functionality, which makes them a solution candidate for a specific task or not. And second, within a configuration of interactions as suggested by an ADP, these solution candidates or respective solution processors take over particular roles for accomplishing specific tasks. That is why different ADP represent configurations (software architecture designs) of computation components (processors), which are likewise suitable (alternatives) for solving the same tasks. It depends on the identification and mapping, i.e. the well-directed allocation of tasks to their processors, which makes best practices utilizable and thus enables to benefit from strategic reuse.

Ward and Mellor show by the example of a bottle-filling system [219, pages 20ff], that different alternatives exists (cf. [219, figures 3.4 and 3.5]) for allocating the features as defined in the essential model to the processors, which make an implementation model. Their proposed allocation process is hand-made and guided only by some recommendations on how to execute "reorganizing the content of the essential model to reflect the choice of a processor configuration" [219, page 36].

By means of solution templates as developed next, this allocation process becomes a patternled one. That is, transition templates are the key to make problems reproducibly absorb into platform on the basis of patterns.

The following sections detail, how to grasp the computational roles and the interactions they provide inside an ADP, and to represent these in a way, i.e. by means of solution templates, which eases the allocation of desired software functionality to a software architecture to-be. Afterwards, this approach is applied to several ADP, such as Client-Server, Forwarder-Receiver, etc. for illustrating its use.

About solutions, and their types and flow inbetween of involved processors

Solution templates are configuration patterns, which emulate the assembly of interacting computational components, which are present in an architectural design pattern (ADP). Solution templates are no new ADP. They simply model known ADP by defined units of software functionality, such as represented by problem templates.

The intent of solution templates is to normalize commonly known best practices in software architecture design, which are documented in varying forms and levels of granularity, cf. appendix D. This is to the advantage of exploring alternative solutions to recognizable problems in a more seamless way, and eases integration of desired software functionality to a software platform to-be.

This approach contributes to brigde the problem-solution gap on a pattern basis, cf. figure 7.2 on page 118. Problems (tasks) and their proper solutions (processors) become identifiable with respect to the same understanding of what are "basic units of activities" that form implementable, and desired software functionality.

Software architecture solutions are compositions of different computational components, which share responsibility for operating tasks. In consequence, only parts of a solution design may be involved in processing a task. These parts take over a role in the task processing, which is nameable, and relatable to specific activities. The intent of using roles in the following is to map recognizable responsibilities to recurring units of software functionality, such as documented in problem templates.

Solution templates illustrate how an architectural design pattern (ADP) implements a defined type of software functionality. For instance, a TOFF-i. solution template presents how a TOFF-i. problem process is mapped to and processed by the respective architectural design pattern.

The following **heuristic** is applied to a selection of ADPs and results for each three solution templates, which are specific to (operate) one type of functionality (TOFF-i. to -iii. problem template).

1. **Pick an (architectural) design pattern to be represented as solution template.**

   Appendix D provides an Overview on Architecture Design Patterns from which a choice is made.

2. **Identify the computational components involved with the pattern.**

   Architectural design patterns are modeled by different means, which are often boxes-and-lines notations or UML sequence diagrams. Boxes and object life lines are first class instances that can be interpreted as computational components.

3. **Model each role as a problem template.**

   Often the names given to boxes or objects are related to and thus applicable as role names. In the following, it is tried to relate these to one out of the three problem templates available, cf. section 7.4, in order to form comparable units of activities for each of these roles.

4. **Assemble roles for designing a solution processor.**

   If each identified computational component takes a specific role, which is representable by one problem template, then these roles can be composed to one solution processor, which emulates the processing, i.e. the interactions of computational components, as defined in the architectural design pattern under investigation.

   According to the classification provided by the second column in table 7.3, the resulting solution template (and its underlying design) represents a pattern (of a solution processor), which has architectural significance.

5. **If a role does not map a problem template, try to make use of it for designing parts of a solution processor.**

   If the scope of responsibility of a role is limited to parts of processing a task, namely to the implementation of its individual activities, then no solution template can be created, and the underlying design is categorized as having no architectural significance, because it does not address the architecture at a coarse-grained design level according to the classification provided by the third column of table 7.3. As the implementation of individual activities is about the actual realization of information storage[11], data transformations, or control transformations, there can be patterns which provide design alternatives to this component level of an architecture (fine-grained design), which are worth to be documented too, but this level is not seen as architectural significant in table 7.3.

Section 7.5.2 presents solution templates for a Client–Server architecture design following the 5-step heuristic as presented on page 137. Section 7.5.3 shows solution templates, which are built based on a Forwarder–Receiver design pattern. Section 7.5.4 embeds an Observer- [90] and a Publish-Subscriber-[46] design pattern to solution templates for TOFF-i., -ii. and –iii. problems. Section 7.5.5 merges the aforementioned solution templates to one architectural blueprint that follows the Model–View–Controller style.

Thereby, it is not only shown how coarse-grained and fine-grained design patterns become classifiable into defined categories. The proposed, pattern-based approach also illustrates that unifying

[11]The patterns *Block Storage* and *Blob Storage* in appendix D.4 provide design alternatives for implementing the data storage (which is only a part) of a computation component according to table 7.3

the units for considering some software functionality in problem analysis and solution design, enables the generation of development alternatives by reuse, and the integration of known problems to best practices solutions.

Each solution template shows, how a problem is (distributed along several computational components of an architectural design, which make it) absorb into a solution (platform).

7.5.2. Solution Templates for Client–Server

The solution templates are developed by following the 5-steps heuristic as introduced on page 137.

1. **Pick an (architectural) design pattern to be represented as solution template.**

   The solution templates given in figure 7.15, figure 7.16, and figure 7.17 give an interpretation of how the **Client–(Dispatcher)–Server** [46, page 323] design pattern as introduced by Buschmann et al. can be used for operating different types of software functionality, namely those as present to the three problem templates designed in section 7.4.

2. **Identify the computational components involved with the pattern.**

   There are two computational components of most relevance to the **Client–(Dispatcher)–Server** pattern, of which one takes the role of a $client$, and the other takes the role of a $server$. Both roles participate, i.e. share responsibility for processing a service (task). The $client$ request services from the server. The $server$ is in charge of all resources required for providing the respective service.

3. **Model each role as a problem template.**

   The $client$ has no computational resources and responsibilities. Client's data and control transformations simply pass over their $trigger$ and $data$ to the server. That is why in each **Client–(Dispatcher)–Server** solution template, both transformations of the client are colored black. Nevertheless, the $Client_{pt2}$ represents an instance of a TOFF-ii. problem template ($pt2$), whose control transformation is in general concerned with the processing of events, and whose data transformation is solely concerned with the processing of data information.

   The $server$ has all computational resources and responsibilities. The problem process (task) is completely executed by this computational component.

   In figure 7.15, the $Server_{pt1}$ executes the control as well as the data transformation, of which both are necessary for operating a TOFF-i. problem process ($pt1$). The server holds and utilizes all computational resources involved with the $result$, i.e. data storage (state).

   In figure 7.16, the $Server_{pt2}$ executes both control and data transformation necessary for a TOFF-ii. problem process ($pt2$). The server provides the $result$ as a reply to the client.

   In figure 7.17, the $Server_{pt3}$ executes both control and data transformation necessary for a TOFF-iii. problem process ($pt3$). The server triggers a new $action$ and respective $result$, which is sent outside the scope of responsibility of the server role.

4. **Assemble roles for designing a solution processor.**

   Each solution template models interactions of a $client$ and a $server$ role to form a solution processor, whose inner organization implements a **Client–(Dispatcher)–Server** software architecture design.

5. **If a role does not map a problem template, try to make use of it for designing parts of a solution processor.**

   In figure 7.15, the data transformation and storage represent one part of the computational component $Server_{pt1}$. This part (and not the entire solution template) can be implemented at a fine-grained level of detail by (architectural) design patterns, e.g. such as Block/Blob Storage, or Relational Database, cf. table D.4 of the appendix D. According to the classification as defined in this work by table 7.3, these ADP, which address a part of a computation component only, have no architectural significance. These kinds of design patterns do not belong to the class of architectural design patterns or styles.

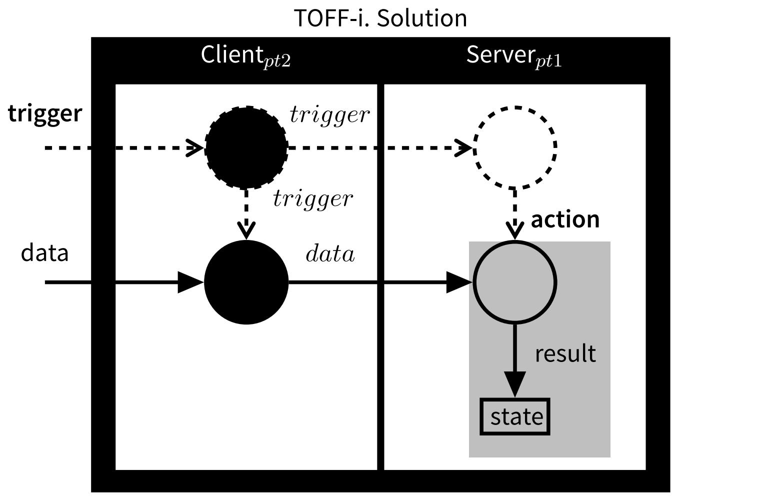


FIGURE 7.15 Client–Server solution – Template for configuring a TOFF-i. processor

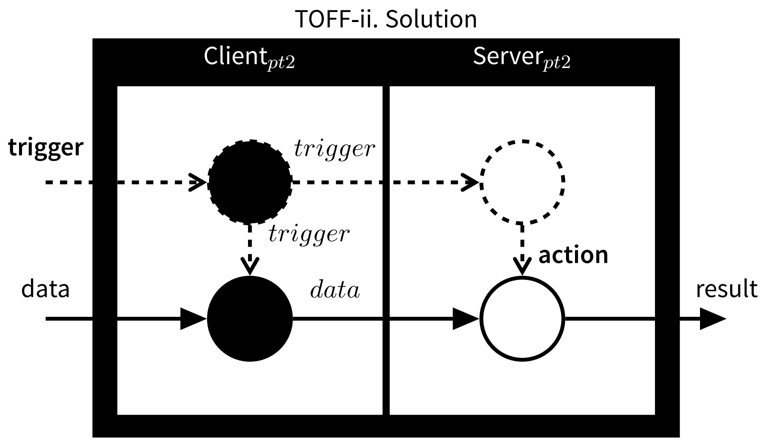


FIGURE 7.16 Client–Server solution – Template for configuring a TOFF-ii. processor

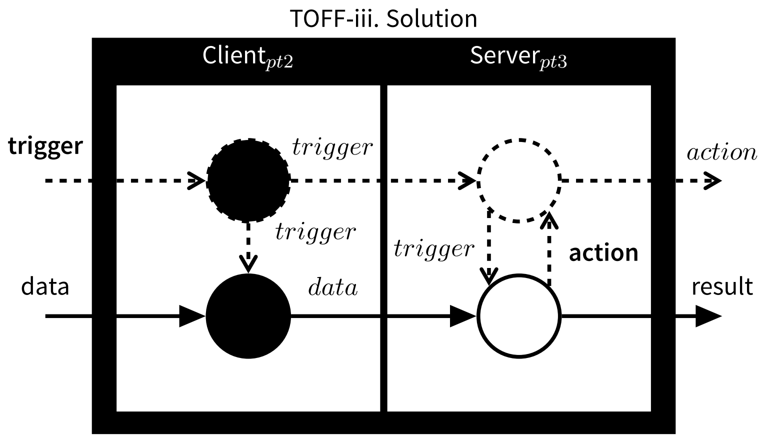


FIGURE 7.17 Client–Server solution – Template for configuring a TOFF-iii. processor

7.5.3. Solution Templates for Forwarder–Receiver

The solution templates are developed by following the 5-steps heuristic as introduced on page 137.

1. **Pick an (architectural) design pattern to be represented as solution template.**

   The solution templates given in figure 7.18, figure 7.19, and figure 7.20 give an interpretation of how the **Forwarder–Receiver** [46, page 307] design pattern as introduced by Buschmann et al. can be used for operating different types of software functionality, namely those as present to the three problem templates designed in section 7.4. The underlying assumption for the interpretation of this pattern is, that a peer has completed a service and invokes its $forwarder$ component to submit the result to the $receiver$ component of another peer.

2. **Identify the computational components involved with the pattern.**

   There are two computational components of most relevance to the **Forwarder–Receiver** pattern, of which one as to be expected takes the role of a $forwarder$, and the other takes the role of a $receiver$. Both roles participate in completing one problem process. They share responsibility for operating a task (service), and reside in one peer. The $forwarder$ completes the service. The $forwarder$ knows the service outcome's receivers and delivers the computational result to these. The $receiver$ is a (kind of dummy) interface, which simply acknowledges the receipt of the result by proxy to its peer.

3. **Model each role as a problem template.**

   The interpretation of the $forwarder$ and $receiver$ roles given here, is inverted to that of the $client$ and $server$ roles as discussed in the previous section 7.5.2. The $forwarder$ is an instance of a TOFF-ii. problem template ($pt2$), and responsible for operating both, the control as well as the data transformations. Thus, it holds all computational resources and responsibilities regarding the execution of a service, which is comparable to the role of a $server$ in the previously discussed ADP. The $receiver$ simply accepts the result computed by the transformations of the $forwarder$. This makes the $receiver$ role comparable to the role of the $client$ as discussed in the previous ADP of Client-(Dispatcher)-Server. That is why in figures 7.18, 7.19, and 7.20 both transformations of the $receiver$ are filled in black. In each of the following solution templates, the $forwarder$ is extended by a data storage $receivers$ (or capable of obtaining this information) , which is accessed by the $forwarder$'s data transformation for addressing the result of a service after its completion, respectively.

   In figure 7.18, the $forwarder$ is in complete charge of operating TOFF-i. problem processes. The $receiver$'s only responsibility remains to make the delivered result persistent, i.e. to make a change to the $state$, but the $receiver$ is not in charge of computing the result.

   In figure 7.19, the $forwarder$ is entirely responsible for executing TOFF-ii. problem processes. The $receiver$ pipes the $result$ computed by the $forwarder$ to its peer.

   In figure 7.20, the $forwarder$ executes both control and data transformations necessary for processing one TOFF-iii. task. The $receiver$ makes the $result$ in the context of a particular $action$-event available to its peer, i.e. it announces the arrival of respective $data$ by this event to the receiving peer.

4. **Assemble roles for designing a solution processor.**

   Each solution template gives the interactions of $forwarder$ and $receiver$ roles, which implements a **Forwarder–Receiver** pattern at the coarse-grained level of solution design, and thus classifies it as pattern of software architecture design according to table 7.3 on page 135.

5. **If a role does not map a problem template, try to make use of it for designing parts of a solution processor.**

   Both roles are applicable to model a solution template. Thus, this step can be skipped.

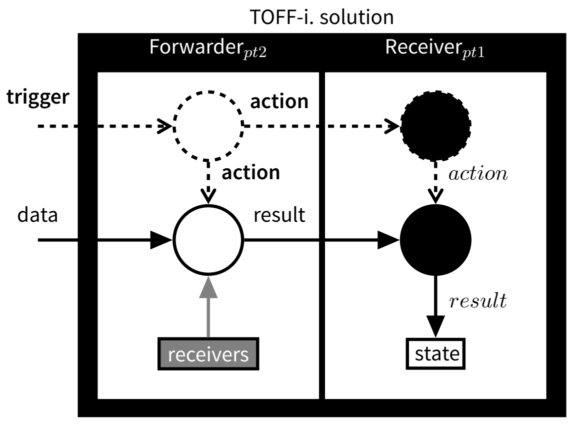


FIGURE 7.18 Forwarder–Receiver solution – Template for configuring a TOFF-i. processor

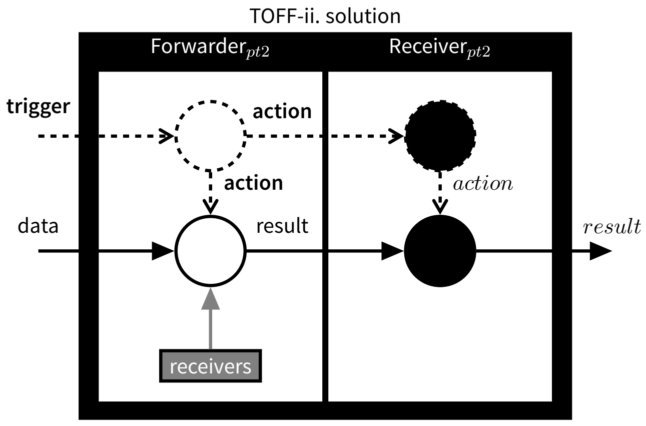


FIGURE 7.19 Forwarder–Receiver solution – Template for configuring a TOFF-ii. processor

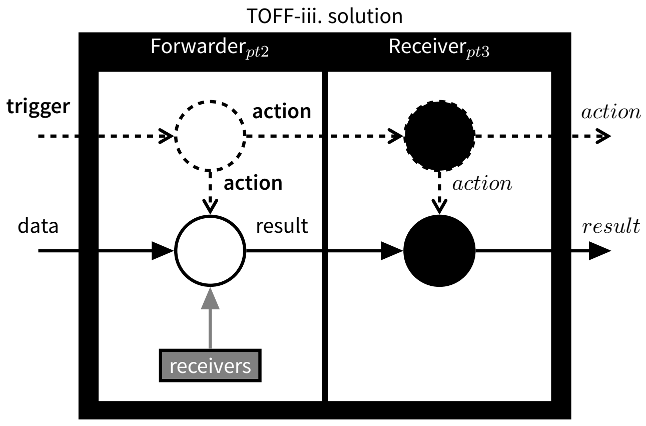


FIGURE 7.20 Forwarder–Receiver solution – Template for configuring a TOFF-iii. processor

7.5.4. Solution Templates for Observer/Publisher–Subscriber

The solution templates are developed by following the 5-steps heuristic as introduced on page 137.

1. **Pick an (architectural) design pattern to be represented as solution template.**

   The solution templates given in figure 7.21, figure 7.22, and figure 7.23 give an interpretation of how the **Publish–Subscriber** [46, page 339] design pattern as introduced by Buschmann et al., which is also known as Gamma et al.'s (GOFs) Observer [90, page 293] design pattern, can be used for operating different types of software functionality, namely those as present to the three problem templates designed in section 7.4.

2. **Identify the computational components involved with the pattern.**

   There are two computational components of relevance to the **Observer/Publish–Subscriber** pattern, of which one takes the role of a $publisher$, and the other takes the role of a $subscriber$. The $publisher$ (or GOF's subject-role) notifies the $subscriber$s about changes (as initiated by the occurence of respective $trigger$-events). Each notified $subscriber$ (or in GOF's terms the observer-role) executes respective $action$s in the context of this announced change (event).

3. **Model each role as a problem template.**

   In each solution template for this pattern, the $publisher$ implements a TOFF-iii. problem template ($pt3$). Accordingly, its data transformation takes over the responsibilities of the control transformation, in order to notify relevant $subscribers$ about which $action$ to execute. The $data$ in itself remains unprocessed by the $publisher$[12]. That is why the data transformation is filled out black. The $publisher$ role is not responsible for computing a $result$. It is concerned with processing control information. In each solution template, the $subscriber$ role receives an $action$-event with respective $data$ from the $publisher$. Its control transformation simply activates the subscriber's data transformation by the received $action$, which is why the control transformation is filled out black. The $subscriber$ role is responsible for computing a $result$. Its main concern is to process data information. The $subscriber$'s data transformation is responsible for processing the received $data$ and producing a $result$.

   Compared to the previously discussd **Forwarder–Receiver** solution templates in section 7.5.3, the $publisher$ is comparable to the $forwarder$, but without the processing responsibility for the $data$. The $subscriber$ is comparable to the $receiver$ role, which in contrast to the $receiver$ represents no dummy interface, since its responsibility is to execute the data transformation.

   Figure 7.21 shows the inner organization of a solution processor, which is capable of operating TOFF-i. problem processes by following a **Observer/Publish–Subscriber** architectural design.

   Figure 7.22 gives a solution template that configures a TOFF-ii. task processor based on $publish$ and $subscriber$ interactions.

   In figure 7.23, the **Observer/Publish–Subscriber** design pattern models the transformations required for implementing a TOFF-iii. problem processor.

4. **Assemble roles for designing a solution processor.**

   Each solution template gives the interactions of $publisher$ and $subscriber$ roles, which implements a **Observer/Publish–Subscriber** solution design, and thus is classifyable as pattern of software architecture design according to table 7.3 on page 135.

5. **If a role does not map a problem template, try to make use of it for designing parts of a solution processor.**

   The identified roles can be combined to one solution template, which makes this step obsolete for further investigations of this software architecture design pattern.

[12] The data processing of the $publisher$ according to TOFF-iii. deals with the choice of a subscriber, who wants to receive the $data$ from the $publisher$.

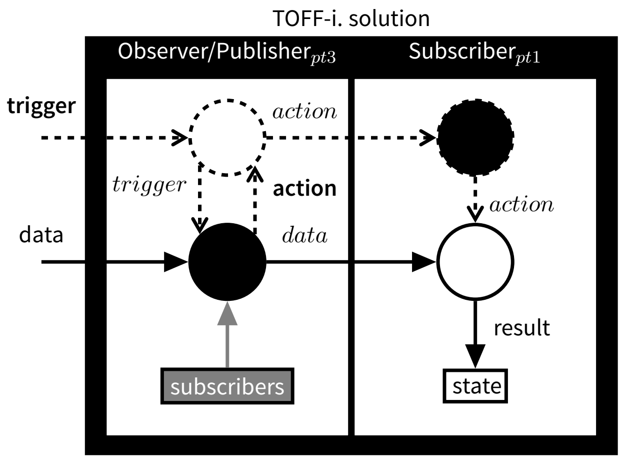


FIGURE 7.21 Observer/Publisher–Subscriber solution – Template for TOFF-i. processors

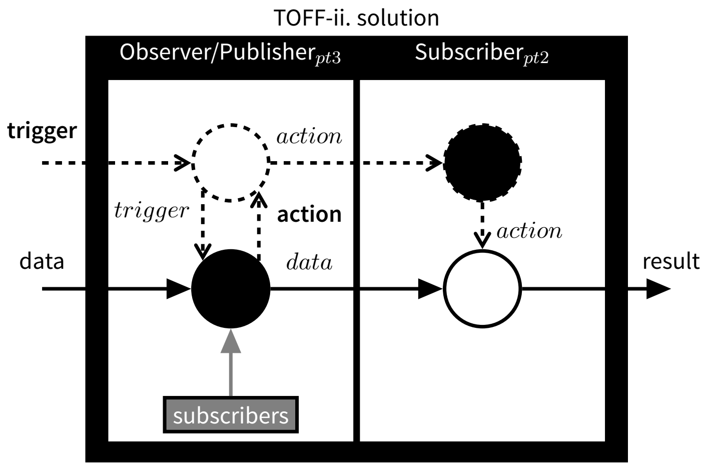


FIGURE 7.22 Observer/Publisher–Subscriber solution – Template for TOFF-ii. processors

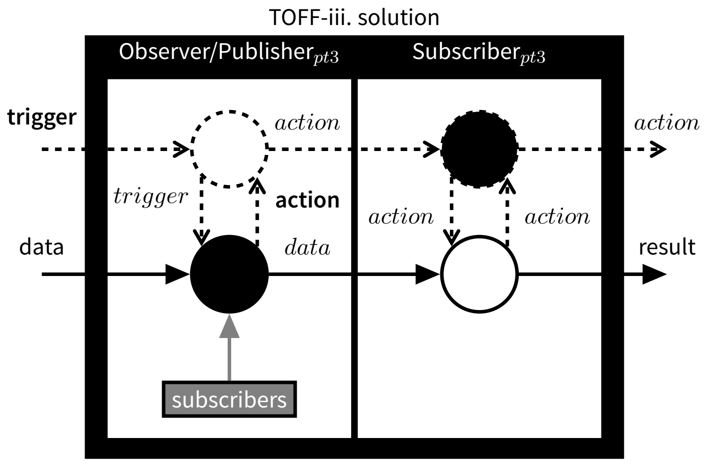


FIGURE 7.23 Observer/Publisher–Subscriber solution – Template for TOFF-iii. processors

7.5.5. Solution Template for Model–View–Controller

The solution template in figure 7.24 and its continued illustration in figure 7.25 are developed by following the 5-step heuristic as introduced on page 137. It combines the solution templates for Client–Server, Forwarder–Receiver, and Observer/Publish–Subscriber as discussed in the sections before into one Model–View–Controller solution design.

1. **Pick an (architectural) design pattern to be represented as solution template.**

   The solution template presented in this section, gives an interpretation of the **Model–View–Controller** [46, page 125] design pattern (MVC) as known from Buschmann et al. by taking the following description into account: *"The MVC pattern is an architectural pattern that leads to a functional decomposition into three groups of components: the Model, the View, the Controller. The Model components represent application data and data access, the View components render data for the user, and the Controller components handle user inputs. The majority of modern Web Applications are based on this pattern; however most Web Application frameworks (e.g., the PHP Zend Framework) implement a slightly modified version of the MVC, the so-called "Web MVC" or "Model 2" pattern [...] In the original MVC pattern, the View is an Observer of the Model, and thus receives direct notifications when changes in the Model's data occur. On the other hand, in the Web MVC version, these notifications are redirected through the Controller instead, as this is easier to realize with many Web technologies [...]"* [89, 157, page 234]. Due to this particular utilization of the Controller, the architectural design as elaborated in the following emulates the **Web MVC** pattern, which supports the processing of each problem template as designed in section 7.4.

2. **Identify the computational components involved with the pattern.**

   There are three computational components that share processing of service (task)s in the **Web MVC** pattern, namely the roles of $model$, $view$, and $controller$. Recapitulating the description of these roles as given above:

   The $model$ cares about data processing and persistence, which relates to TOFF-i. software functionality as is implemented by the data transformation of the $Subscriber$, see 3. in figure 7.24.

   The $view$ cares about the processing of data (to obtain a result) that is to present to a user or an entity external to the web application. This kind of behavior or computational responsibility relates to TOFF-ii. and TOFF-iii. software functionality as is implemented by the data transformation of the $Subscriber$, see 4. and 5. in figure 7.24.

   The $controller$ is as an intermediary always involved with the processing of $model$ as well as $view$, which must be taken into account by the solution template to be modeled next. The respective responsibility of the $controller$ role is implemented by the $Observer/Publisher$, see 2. in figure 7.24.

3. **Model each role as a problem template.**

   The **Web MVC** pattern in figure 7.24 and as continued in figure 7.25 is the starting point for the Client–Server solution as developed in section 7.5.2. The computational component $Server_{pt1}$ in the problem template of figure 7.15 provides a TOFF-i. solution, which takes over the role of the $model$ as defined for by the **Web MVC** pattern. The $Server_{pt2}$ in figure 7.16 provides a TOFF-ii. solution, the $Server_{pt3}$ in the problem template of figure 7.16 provides a TOFF-iii. solution, which both take over the role of a $view$ as defined by the **Web MVC** pattern. That is, why the figure 7.24 and figure 7.25 are vertically structured into a $client$-part and a $server$-part (column), and horizontally into three sections (rows), which each addresses the processing of a specific problem template. These computational components represent jointly the solution template for the **Web MVC** pattern.

   The $Server$-part in figure 7.24 and figure 7.25 is further refined by use of the Observer/Publish–Subscriber solution templates as developed in section 7.5.4 to take the role of the $controller$ into account. As a result, each $Client$-request (see 1. in figure 7.24) to the $Server$ passes the Observer/Publisher-component. Each $trigger$-event, which initiates the operation of one specific type of software functionality, is redirected via this computational component, which is central to the **Web MVC** pattern. This is also the reason for the gray-shading of the computational components above and below the $Controller$ 2.. There is only one $Controller$ component (in the center of figure 7.24), which manages all $Subscriber$s and thus controls the $model$ and $view$s respectively.

   Figure 7.25 shows only for the sake of completeness, that the $Subscriber$-components are refineable by means of the Forwarder–Receiver pattern as discussed in section 7.5.3 to manage the communication of reply messages from the server to the client. This detail is out of scope for the **Web MVC** pattern, and thus not considered further.

   In figure 7.24, the TOFF-i. $Subscriber$ 3. is in charge of all computational resources to fulfill the role of the $model$. It processes received data and produces a result, which is stored by its data transformation. The result remains in the scope of responsibility of the $Server$.

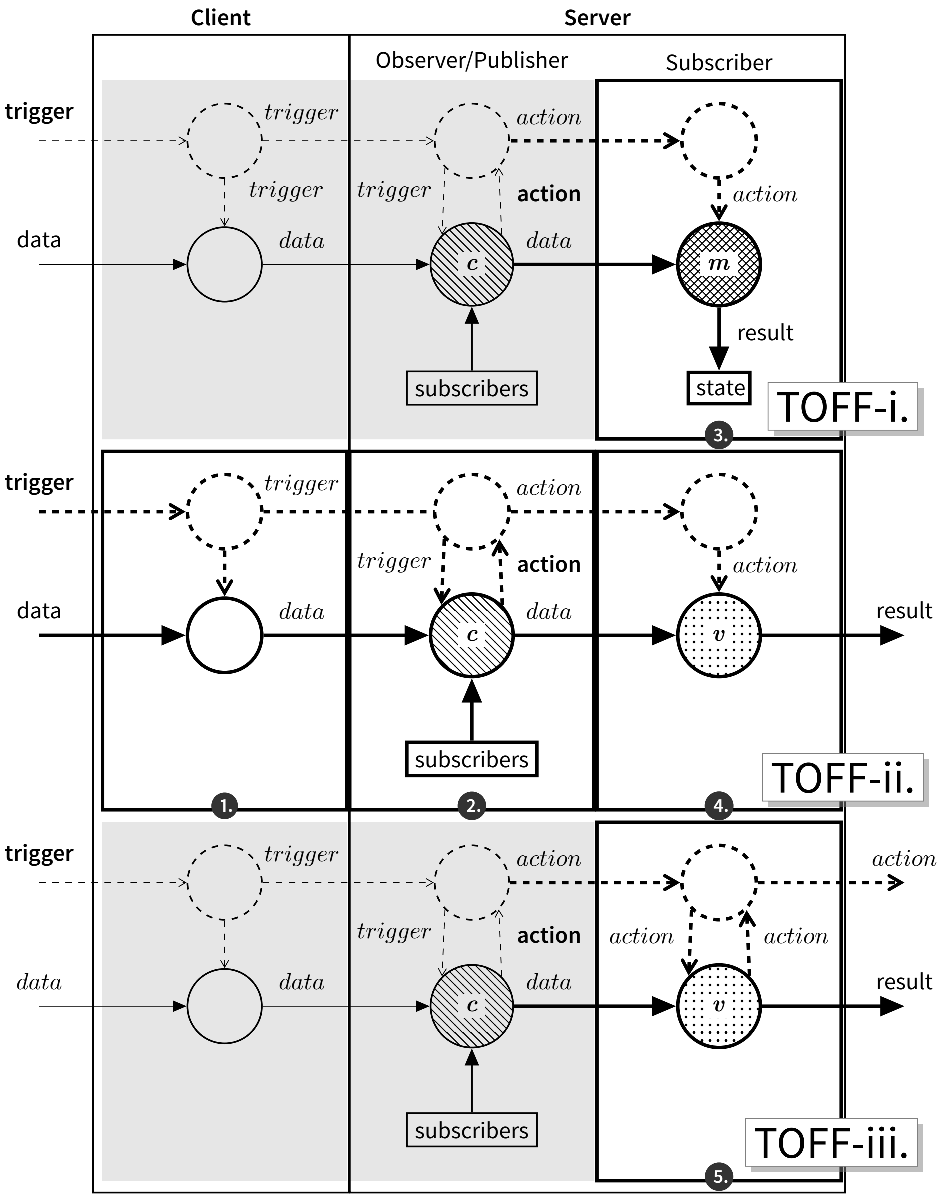
   The TOFF-ii. 4. and TOFF-ii. 5. $Subscriber$ process data information, but send the calculated result back to a $Client$. These two computational components take the role of a $view$. An announcement-$action$ received from the $controller$ provokes each $view$ to update its status or respective presentation, i.e. to start the processing and involved delivery of some information.

4. **Assemble roles for designing a solution processor.**

   The bold emphasized interactions and involved computational components of $model$, $view$, and $controller$ in figure 7.24 form a joint solution processor, which is structured according to the **Web MVC** pattern. Its collaborating, computational roles share the responsibility for (the processing of) basic units of activities (tasks). According to table 7.3 on page 135 this pattern belongs to the level of coarsed-grained solution design as used to model software architectures.

5. **If a role does not map a problem template, try to make use of it for designing parts of a solution processor.**

   The identified roles can be mapped to task patterns and combined to one solution template, which makes this step obsolete for further investigations of this software architecture design pattern.



m data transformation takes responsibilities of role *model*

c data transformation takes responsibilities of role *controller*

v data transformation takes responsibilities of role *view*

Box gray-shading masks negligible computational components, whose redundant reference results only from assembling these to one solution template

FIGURE 7.24 Solution template for Web Model–View–Controller (part 1, Client-to-Server)

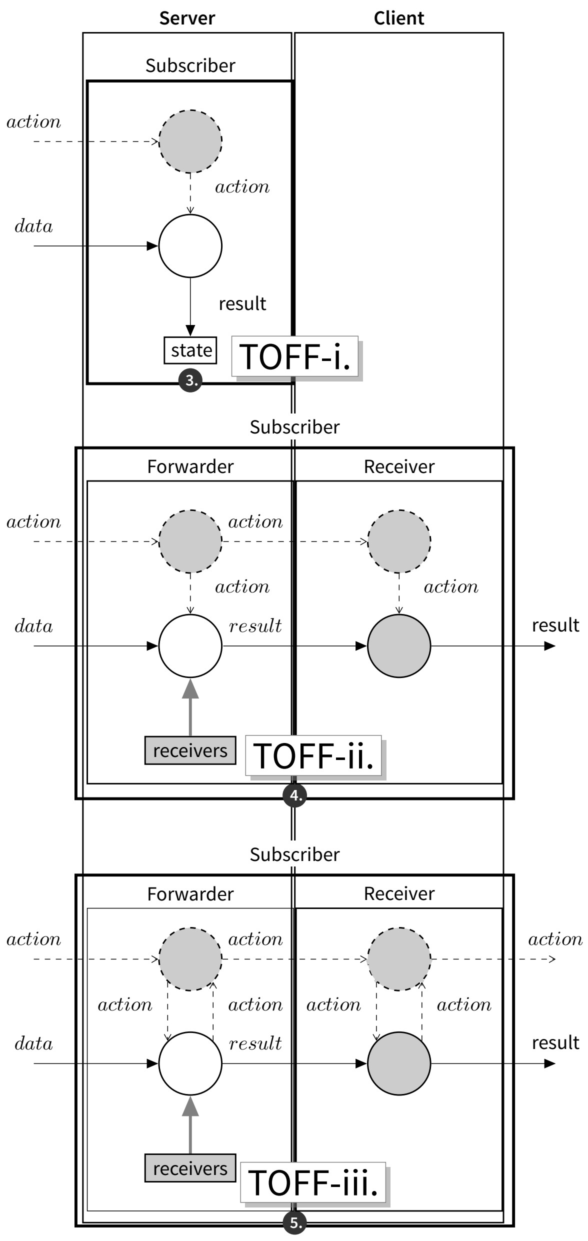


FIGURE 7.25 Solution template for Web Model-View-Controller (part 2, Server-to-Client)

7.6. Discussion & Related Work

Problem-based units of work as developed in this chapter result from an instant mapping of emerging problems to candidate solutions, which is led by patterns. According to Meyer (2014), the "choice of abstraction is essential." [156, page 112] for avoiding a design vision, which yields a hardly refactorable software architecture in the presence of changing requirements.

By making patterns of problem analysis and solution design account for the concept of "basic units of activities", a constant level of granularity for looking at desired software functionality and the options available for its fulfillment becomes available. This eases the allocation of requirements to a software platform to-be, and makes related decision-making transparent. Transition templates as have been elaborated in this chapter address this need, and thus contribute to the co-development of requirements and software architecture.

For this, they not only develop the findings of part II of this work further by making use of therein introduced units for reproducibly scoping software requirements. Transition templates also constitute the fundamentals for a role-driven mapping of pattern-based best practices knowledge, which has been started in the work of Schmidt and Wentzlaff [191] for implementing a quality-aware and pattern-oriented analysis and design approach, whose advancement in Côté et al. [69, 70] has received noticeable attention.

Transition templates prove Alebrahim [3] wrong in her review of Schmidt and Wentzlaff [191], that "there is no systematic approach given for selecting appropriate architectural patterns" [3, page 350]. Any architectural pattern is suitable, as long as it is known, how its interacting computational roles share the processing and thus responsibility for a defined task. This knowledge is grasped by transition templates.

Nevertheless, the research as discussed in here is in accordance with Buschmann and Henney, who assert that "Problem Frames are an important vehicle to get first hands on a software architecture." [45, slide 19].

While Alebrahim [3, page 149, step 2] proposes in phase 2 of the Quality-based Co-Development of Requirements and Architecture (QuaDRA) framework the use of a pattern catalog and respective questionaire, whose development and application requires massive upfront work in addition to the work required anyway for addressing the actual problem model at hand; transition templates benefit from the (frame) structures, which are already in place, and makes them accessible in an instant fashion. Instead of artificially reducing the design space, possible solutions relate naturally to the details given in the problem description.

Rapanotti et al. [99, 182] envisioned a bridge between requirements and software architecture on the basis of patterns by introducing Architectural Frames (AFrame). These are problem frames, whose domains are labeled with role names, which are taken from descriptions of commonly known architectural styles, such as Pipes-and-Filters, or MVC. The use of AFrames creates a solution-aware decomposition of problems given in frame diagram notation. A frame concern serves as correctness argument for maintaining the composition, i.e. the dependencies of problems that belong to one AFrame. In a worst case, this approach increases the complexity of a requirements model depending on the number of problems that define the composition of one AFrame.

Choppy et al. [50] defined for problem frames a structural mapping to yield solution designs, which follow a layered architectural style and are denoted by UML composite structure diagrams. Problem dependencies are maintained by state machines, which model the application's behavior. In contrast to AFrames, problem decomposition by architectural patterns for problem frames does not affect the requirements model in itself. These patterns care about separating the problem from its solution. Problem dependencies on architectural components (are handled by merge rules, which) guide the integration and respective composition of pattern-based solutions to one combined architecture and software life-cycle.

Agility in software projects demands the "ability to keep adjusting the product to emerging needs through the addition of new features" [44, page 12]. In this context, the "easy composition of software assets increases in importance" [37, page 1].

In order to meet this concern, "Align feature-based development and system decomposition [represents one architectural tactic that merges in an ...] architectural runway" [23, page 21], which helps the project team "in forming a mental image of the desired system" [144, page 199], and that way makes hidden assumptions and architectural dependencies visible.

The application of an intentional architecture [144, chapter 16] is a key factor to successfully scale agile software development projects. It "provides the guidance needed to ensure [an enduring design, i.e.] that the whole system has conceptual integrity and is fit for its purpose" [189].

The availability of an architectural blueprint makes a degree of "stability [available, which is] required to support [the planning of] development [, and] is particularly important to the successful operation of multiple parallel [...] teams" [23, page 22] for projects at scale. These "need to be as decoupled in their work as possible" [37, page 2] "to allow independent decision-making and reduce communication and coordination overhead" [23, page 22]. For planning "stories to be developed within each [project] iteration, the team identifies [based on this blueprint] the architectural elements that must be implemented" [44, page 13] to satisfy these.

Transition templates and architectural patterns for problem frames comparably support the modeling of a blueprint that demonstrates the allocation of desired software functionality to an intented software platform, and accounts for involved architectural dependencies, even though by utilizing different notational means.

In doing so, transition templates implement a feature-driven, vertical slicing approach, whereas architectural patterns for problem frames operate in an infrastructure-driven, horizontal slicing[13] mode [23, figure 2, page 21], which supports in both cases based on patterns

- the decomposition of problems, i.e. their refactoring[14], to fit an architecture design.
- the composition of a software design, i.e. the integration[15] of computational components into one combined architectural solution.
- the management of functional dependencies, i.e. their proper separation[16], which is made feasible by taking the software application's life cycle into consideration.

Both approaches keep a problem description untouched for exploring blueprints of a software system to-be. Their utilization of patterns establishes a means for Architecture Agility by "informed anticipation", which "allows architectural development to follow a "just-in-time" model[, that] maintains a steady and consistent focus on continuing architectural evolution in support of emerging customer-facing features" [44, page 12].

Problem-based units of work are of twofold use to project planning. This chapter 7 introduces how the software engineering role in a project team benefits from transition templates to design a problem-based unit of work, which outlines what is to be done for satisfying user expectations. The following chapter 8 details how the team can face their project management responsibility, by taking advantage of transition templates to decide on and justify, what problem-based unit of work becomes part of the plan for the next project iteration.

[13]system decomposition/view on a software architecture
[14]restructuring without change of external behavior
[15]bring together parts into one whole
[16]reduction of complexity by a proper scoping of software functionality to self-contained, recognizable units

7.7. Summary

This chapter introduces transition templates, which are pattern of patterns to “bridge the intradomain semantic gap” [53, page 33] between problem analysis and solution design. They are built on a customization of Ward and Mellor’s transformation schemata for identifying those “basic units of activitites”, which are inherent to patterns in both domains, thus establishing a link between these.

As today’s “most software development is more concerned with composition of existing [...] components in creative configurations” [37], rather than “creating masterpieces from nothing” [159, page 11], it is indispensable to unlock the “body of software practice experience” [135, page 4], which “engineering have evolved over time [as] a collection [...of patterns]”. These collections enshrine “established, shared understanding of the common forms of design” [198, page 19], which “serve as a shared, semantically rich vocabulary” [198, page 2] that belongs to “one of the hallmarks of a mature engineering field” [198, page 19].

For this purpose, transition templates determine recurring computational responsibilities as are generally defined and structured by patterns of software analysis and design, and which frame the inner organization of software requirements models, as well as of models for software architectures.

Problem templates as designed in this work are special transition templates to tailor requirements to tasks, which represent a machine interface specification that possess a recognizable type of software functionality and constant level of granularity. These task patterns build on and advance the findings, which have accompanied the development of problem-based functional size measurement patterns in part II of this dissertation.

Solution templates are designed in this work to configure the computational components that form a software’s architecture design, into a solution, which is capable of processing tasks as defined by problem templates. These configuration patterns exhibit the different roles and interactions, which are taken over by the components that share responsibility for processing a task. Solution templates emulate commonly known architectural styles and design patterns as listed in the appendix D.

That way, transition templates guide the allocation of desired software functionality to candidate software architecture designs. By utilizing “basic units of activities” as a boundary object, which provides for unifying the user’s and developer’s understanding of a defined set of software functionality, transition templates bring a project team in the position “to easily incorporate new requirements in the system in a cost effective fashion.” [37].

The application of transition templates to requirements work packages results in problem-based units of work, which are a key enabler to speed up software development and shorten time-to-market. They provide for improved anticipation of requirements-related adoptions by an early envision of independent engineering effort at which project “teams can perform their work with minimal dependency on other teams.” [37]

Problem-based units of work implement an approach to “Requirements engineering[, which is...] more tightly integrated with system implementation to take advantage of reuse and to let systems evolve to reflect changing requirements. [...This] is the best hope we have for more effective software engineering” [201, page 23].

8. Problem-Based Adaptation Framework

8.1. Introduction

This chapter presents an approach to problem-based project adaptation, which copes with the intertwining of requirements and architecture based on pattern practices, and focuses on dependency issues involved with these.

It integrates problem-based units of work as developed in the previous chapter 7 into a view model of software architecture, which captures the different perspectives and concerns, the various members of a project team have regarding the software functionality to be built. This description serves the team as a communication model, which not only allows them for establishing but also for reviewing their project planning.

Section 8.2 Background introduces Kruchten's 4+1 View Model on software architecture [133], which provides for organizing the decision-making on the architecture of a software system. This decision-making is accompanied by multifaceted dependency concerns, which at the architectural level are best addressed through considering the software life cycle. For this purpose, different notations to express and approaches to manage it are presented, and developed further in the following.

Section 8.3 All for One and "One4All" – An architectural view model for the three amigos adapts the 4+1 View Model to apply to problem-based units of work. By establishing correspondence among the different views in a pattern-led, and value-driven[1] approach, the resulting "One4All" view model on software architecture comes along with a built-in dependency management for defined units of software functionality, which fast-tracks the team in switching priorities on demand for an effective project planning.

Section 8.4 Problem-Based Project Adaptation by the One4All View Model summarizes which view addresses which concern, and how the means developed in this work contribute to it.

The focus is on managing the dependencies between requirements and architecture. That is why section 8.5 Synchronizing Requirements by a State Transition Diagram uses state transition diagrams in the Process View of the One4All model as alternative to life-cycle expressions. It gives an example of their application to the case study of a Student Recruitment Web Portal, and thereby illustrates the ease of synchronizing the items from the Problem View, namely requirements work packages, into an operational work sequence.

Section 8.6 Sample Application to Use Case Decomposition reasons about a problem-based use case decomposition. It executes UML use cases decomposition by a problem-based requirements analysis for the case study of a Student Recruitment Web Portal. And it adds some rules to guide the creation of use cases on the basis of problem-based functional size measurement patterns. As a result, use cases that share a common level of granularity become obtainable, which are systematically concatenated by state transition diagrams.

In order to gain more insights on the role and importance of architecture to software project adaptation and planning, section 8.7 Discussion & Related Work takes a closer look at the intertwining of requirements and architecture in agile software development projects. In addition, it investigates how problem-based interaction analysis and dependency management relate to each other in this context.

Section 8.8 Summary presents the findings of this chapter, i.e. the dues of having a "big picture" at hand, which makes the identification of decoupled responsibilities and their assignment to respective roles, both in the technology and in the team, possible.

[1]since business process-oriented aka software life cycle-founded

8.2. Background

This section provides the background on an architecture-centric, problem-based planning method, which joins the different perspectives taken by users and developers, who work in one project team, into one shared, pattern-based communication model.

It supports the team in the envisioning of “blueprints” for building the software to-be, but also provides them a pathway to choose among these. This is made possible by explicitly taking requirements dependencies, and their impact on the components of an architecture design into account.

Therefore, the software life cycle is in focus of this chapter. It enables an informed decision-making for the project planning, in regard to which desired software functionality should be done first for satisfying user expectations, but also for skimming the most value out of the aspired technological solution.

Section 8.2.1 The 4+1 View Model on software architecture introduces an architectural framework for organizing the different views of the team members into one comprehensive description. It is adapted in the following to operate on patterns, which not only serve to establish correspondence among the views, but also care for quality assurance in each perspective.

Section 8.2.2 Dependency Management for Problem-Based Units of Work recapitulates approaches and in particular the notations used to manage requirements dependencies, and how these are utilized to assemble a solution design and, in the course of this chapter, how they enable decision-making on an accordant project work plan.

8.2.1. The 4+1 View Model on software architecture

Figure 8.1 gives a slightly consended representation of the "4+1 view model" of software architecture developed by Kruchten (1995), which accounts for the fact, that the various members of a project or software development team have unique concerns involved with (what is) a software's architecture, and respective levels of detail for its consideration. Their different perspectives are commonly referred to as views. Thus, the attempt to gather "the gist of an archictectural design" [133, page 42] in one single boxes-and-lines diagram (architectural blueprint) is a futile endeavor, especially when it serves for communication and review purposes across the different members or respective roles in a team. Kruchten proposes a model, which "describes software architecture using five concurrent *views*." [133, page 42], which a team can use to "organize the description of their architectural decisions around these" [133, page 43]. Figure 8.1 shows, that Kruchten's 4+1 View Model consists of: the logical view, development view, physical view, process view, plus one view for the scenarios. In addition, each is assigned with a respective role in a software project or development team, which is most concerned with it. "Each view is described by [. . .] a "blueprint" that uses its own particular notation" [133, page 43], but notations and tools other than the described can be used just as well.

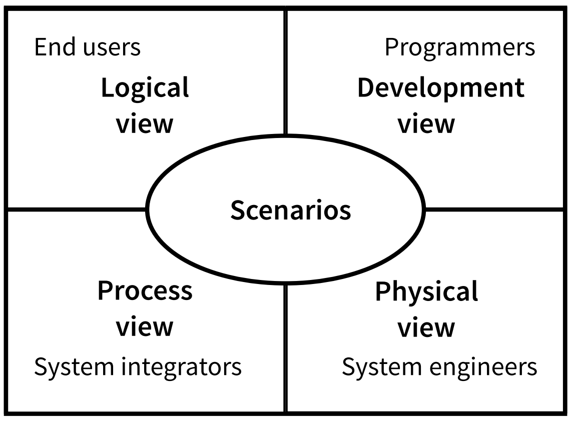


FIGURE 8.1 The 4+1 View Model, slightly adapted from Kruchten [133, page 43, figure 1]

At the heart of the 4+1 View Model are the scenarios, which serve "to show that the elements of the four views work together seamlessly." [133, page 47]. So, it is essential to establish "correspondence among [all] views" [133, page 47]. Kruchten establishes this correspondence through focusing on the identification of "a set of key abstractions" [133, page 44], which are meaningful in the problem as well in the solution domain, and which he denotes as "items that are architecturally significant" [133, page 44].

To this end, the "4+1 view model" makes "The description of an architecture – the decisions made – [transparent to the team, which then] can be organized around these views [. . . a description] which deals with the design and implementation of the high-level structure of the software. [. . .] It is the result of assembling a certain number of architectural elements in some well-chosen forms to satisfy the major functionality [. . . and non-functional requirements] of the system" [133, pages 42, 43].

Logical View

The logical view serves the end user. It "primarily supports the functional requirements – the services the system should provide to its end users." [133, page 43] These have been decomposed into "a set of key abstractions, taken mainly from the problem domain [...] that exploit the principles of abstraction, encapsulation, and inheritance. In addition to aiding functional analysis, decomposition identifies mechanisms and design elements that are common across the system." [133, page 44] The logical view makes use of class diagrams and templates to form desired key abstractions.

Process View

The process view serves the systems integrators. Thus, it "takes into account some nonfunctional requirements [...and] addresses concurrency and distribution [....It] can be seen as a set of independently executing [processes,] which in turn are connected [...] and may exist simultaneously, sharing the same [...] resources." [133, pages 44 and 45]. These processes must be synchronized properly. Therefore, the software is partitioned into a "set of independent tasks; separate threads of control that can be individually scheduled" [133, page 45]. The major challenge in here is to identify the "architectural elements that can be uniquely addressed" [133, page 45], and to let them "communicate through a set of well-defined intertask-communication mechanisms" [133, page 45].

"To determine the "right" amount of concurrency and define the set of necessary processes[,...] classes (and their objects) [are mapped] onto a set of tasks and processes[2]" [133, page 48], which follows the principles of Rubin and Goldberg [186]'s object behavior analysis . To model the "blueprint" for the process view, Kruchten proposes the use of architectural styles from Garlan and Shaw [91], Shaw and Garlan [198] to establish an intertask-communication mechanism of architecturally significant elements.

Development View

The development view serves the programmers. It "focuses on the organization of the actual software modules in the software development environment. It supports the allocation of requirements and work [...,] and reasoning about software reuse [....] It is the basis for establishing a line of product. The development view is represented by [a composition of architectural significant elements to] modules [of software code] and subsystem diagrams that show the system's export and import relationships. " [133, page 45]

Physical View

The physical view serves the systems engineers. It maps the "various elements identified in the logical, process and development views" [133, page 47] onto their processing nodes, i.e. this view accounts for the physical deployment of a software.

Scenarios

A scenario serves to integrate all views. It "acts as a driver to [...] discover architectural significant elements [...,] and it validates and illustrates the architecture design, both on paper and as a starting point for the tests of an architecture prototype. [...] The scenarios are in some sense an abstraction of the most important requirements, each is modeled by corresponding *scripts* (sequence of interactions [...]) as described by Rubin and Goldberg [186]" [133, page 47]. Capturing scenarios as use cases is also a common approach to model this view.

[2]According to Kruchten [133, see URL, page 4], a *process* is a grouping of tasks that forms an executable unit.

8.2.2. Dependency Management for Problem-Based Units of Work

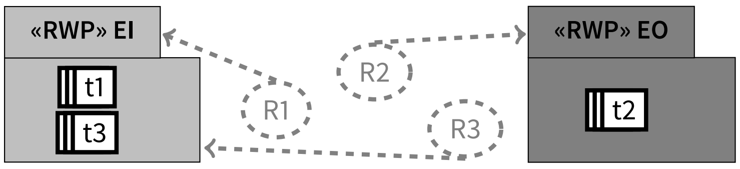
Dependencies of any kind add to complexity and represent a root cause of unwanted interactions, if not dealt with properly. According to Nord et al., "Dependency analysis is used to determine the precedence in the implementation of the features" [160, page 160], and complemented by Rubin and Goldberg, "the order in which activities take place within a system is one of the principle causes of change requests in big systems" [186, page 61]. Thus, it is apparent, that dependency management is cruicial to the success of software projects. As functional user requirements are fundamental to a software project plan, these are the starting artifacts to analyse and classify requirement dependencies, and to introduce, what this dissertation proposes for their management:

- Dependencies among requirements
  - Structural dependencies among requirements are reflected by the allocation of these (in the form of tasks) to the same problem, i.e. to one requirements work package.

    For instance, due to requirements decomposition by use of (functional size measurement) patterns, and the application of problem templates, which are task patterns to streamline the requirements, the resulting tasks $t1$ and $t3$ belong to RWP_{EI} and task $t2$ belongs to RWP_{EO}. RWP_{EI} and RWP_{EO} represent different kinds of problems to be solved.

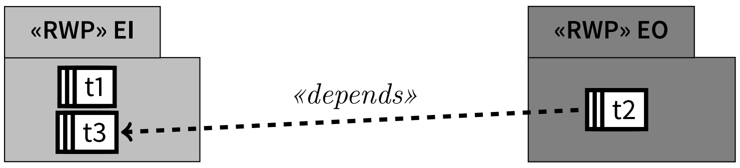
    In consequence, $t1$ and $t3$ $depend\ on$ the same type of desired software functionality as is covered by RWP_{EI}.

    This kind of dependence is intended to limit the complexity of problem descriptions, and to establish a constant level of granularity for considering requirements.



  - Functional dependencies among requirements are reflected by composing respective tasks to one application life cycle.

    To continue the example from above, irrespective of the relation between tasks and requirements work packages, it may be the case, that tasks depend on each other.



    For example, task $t2$ $depends\ on$ events or data produced by task $t3$. This dependence is accountable by constraining the timely order of processing $t2$ and $t3$ through a respective life cycle expression LC, which models the runtime behavior for the software application to-be, such as

    $LC ::= <t3;t2>$,

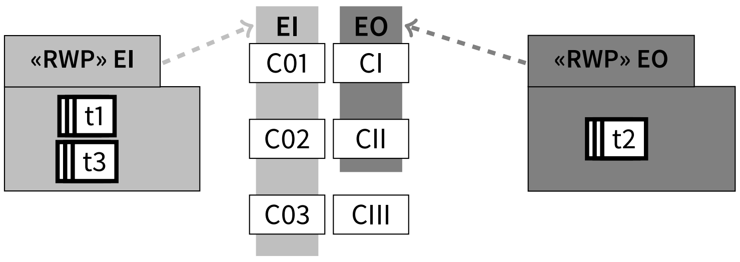
    which defines a sequential relation between these two tasks, and denotes that $t3$ is executed before $t2$.

- Dependencies between requirements and architecture
  - Structural dependencies between requirements and architecture are reflected by the allocation of tasks to the computational components that constitute a software architecture design. This kind of dependency is managable via solution templates as introduced by this dissertation, which are patterns that represent a problem-based configuration of an architecture design.

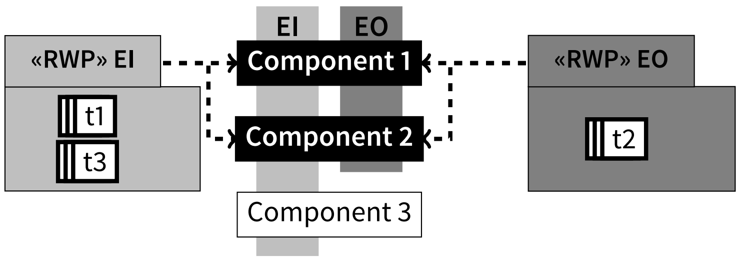
    For instance, by use of solution templates, it is known, how to allocate the tasks involved with RWP_{EI} to the computational components (or vertical slices), namely *C01* to *C03* of an software architecture, which for illustration purposes follows here a layered style.

    In this example, each EI-problem makes use of an architectural component C01, which may represent graphical user interface functionality, an architectural component C02, which can be access functionality to a data base, and an architectural component C03, which controls access rights to the data base. These components share responsibility for a complete execution of EI-tasks. Tasks that belong to RWP_{EO}, such as *t2*, depend on the components *CI* and *CII* only.

    Accounting for this kind of dependence eases the integration of problems to suitable solutions, and to take advantage of reuse.



  - Functional dependencies between requirements and architecture are reflected in the planning of work packages for a software project. They promote the composition of a requirements baseline. This takes into account the component usage as provided in the architectural design for the software application to be built.



The example shows that software functionality as desired by RWP_{EI} and RWP_{EO} *depends on* the same components *Component 1* and *Component 2* given in both architectural designs. That *C01* and *CI*, as well as *C02* and *CII* have been identified as the same architectural components, and that finally two different kind of problems depend on the same architectural components, this synergy (horizontal slice) is again made possible due to the use of patterns for classifying the requirements.

Knowing this dependence affects project planning in several ways. First, it enhances the quality of requirements. For instance, missing requirements or incomplete requirements become identifiable by looking at their utilization of computational components in the architecture. In the given example, a read of the data base (*Component 2* via *t2*) without a preceding write to the data base (an update of *Component 2* via *t1* or *t3*), as well as the omission of *Component 3* by *«RWP» EO* can be checked for plausibility. For instance, *t2* may do without *«RWP» EI*-tasks, because it is in the position to present an empty read result. In this case, the functional dependence still remains, and in addition, there is confidence, that it has no overlooked consequence.

Second, knowing the functional dependence of requirements to architecture helps the planner in identifying hot-spot parts of a software application, which are vital to the fulfillment of many user needs or critical to the maintenance of entire business processes. This issue is addressable by software life-cycle considerations, which are elaborated in this section in depth.

Third, it allows the planner for synchronizing development work with respect to the resources available in team and technology, which in consequence avoids accidential and thus costly replication of already in existence software functionality.

Life-Cycle Expressions

Software Life-Cycle Expression (LCE) belong to the early problem analysis steps of the ADIT software development process as introduced by Heisel and Hatebur [101, 110], which is tought in the lecture Software Technology [108, step A.6, slide 491] at the University of Duisburg-Essen. Their purpose within the ADIT process is to document the relation of software requirements specifications, which are given as UML sequence diagrams, and which have been derived separately during problem decomposition.

In accordance with Rubin and Goldberg, they “typically [model] a time-sequenced ordering of the major activities that occur in the problem domain” [186, page 50]. Software life-cycle expressions describe the "usage protocol" of a software to-be by restricting the order in which these major activities as defined in the specifications may be invoked.

This kind of ordering accounts for the functional dependence of requirements among each other, which in consequence involves related computational components that compose the software’s architecture design. That is why, software life-cycle expressions are applicable for modeling the configuration of computational components and thus serve the composition of an overall software solution. This kind of solution composition is illustrated by the example of a vacation rentals application in [108], which is adopted in section 10 Vacation Rentals Web Application of the Case Studies part V to illustrate conceptual improvements in regard to the software life-cycle as contributed by this work. Côté et al. [70] developed the area of application for life-cycle expressions further by making use of these for the composition of software architecture design patterns, which are used to build a web chat application.

The lecture notes [108, slide 497] define life-cycle expressions according to Coleman et al. (1994) in [61] as follows: They are expressions over the alphabet: $<$*sequence diagram name*$>$. Each *sequence diagram name* is a life-cycle expression. If x and y are life-cycle expressions then

$x;y$	x is followed by y
$x \mid y$	either x or y
x^*	x is executed 0 or more times
x^+	x is executed at least once
$[x]$	x is optional
$x \mid\mid y$	x and y are executed concurrently
Operator precedences:	$[], {}^*, {}^+, ;, \mid, \mid\mid$
Definitions:	*name*[3] = *life-cycle expression*

[3]name can then be used in other life-cycle expressions (but recursion is not allowed!)

State Machines and State Transition Diagrams

Choppy et al. [51, 52] make use of UML state machines in connection with requirements dependencies to manage the composition of components in an architectural design to one overall software solution.

In contrast to life-cycle expressions, state machines have the advantage of making the (pre and post) conditions (in form of states) explicit, that call for the execution of specific software functionality, and which represent anchor points for a problem-based composition of the solution. In addition, they allow for nesting states and assigning multiple meanings to these by separating them into orthogonal regions. This extends the expressiveness of state machine compared to life-cycle expressions.

Ward and Mellor model "externally observable behavior of the system being controlled" [217, page 65] by making use of State Transition Diagrams (STD), which describe and allow for composing control transformations that "map input event flows to output event flows" [217, page 64] in a software system. State transition diagrams use "states to represent intervals in time over which some behavior persists and transitions to represent points in time at which behavior changes" [217, page 68].

Both approaches share, that they define a "finite automaton with output [. . . , which] consists of a set of states[, . . .] an alphabet of input symbols, an alphabet of output symbols, and a transition function that maps combinations of states and input symbols into states[, where . . .] both the input and output alphabet are associated with event flows." [217, page 67] This finite automaton is organized as "a *Mealy machine*[, which] associates each transition (that is, each combination of state and input symbol) with an output symbol." [217, page 68]

The problem involved with both approaches is the risk of increased complexity through state explosion and confusion in leveling these, if the creation and reference of states is not controlled sufficiently. These two problems are not addressed by the concepts used by Choppy et al. and Ward and Mellor, and thus calls in for investigating further means. Otherwise, "the diagram can become hopelessly tangled" [217, page 69].

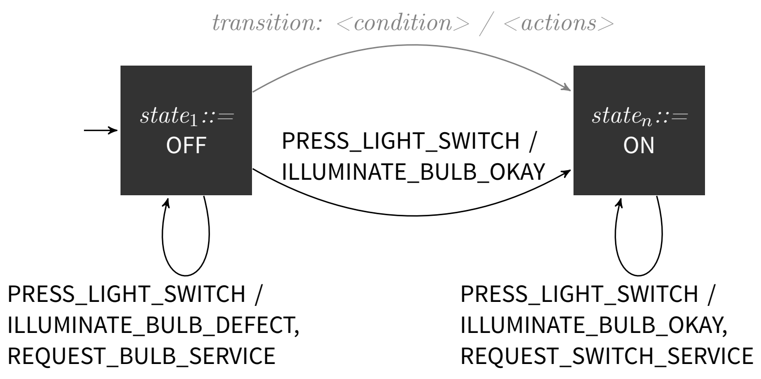

transition: <condition> / <actions>
$state_1$ ::= OFF
$state_n$ ::= ON
PRESS_LIGHT_SWITCH / ILLUMINATE_BULB_OKAY
PRESS_LIGHT_SWITCH / ILLUMINATE_BULB_DEFECT, REQUEST_BULB_SERVICE
PRESS_LIGHT_SWITCH / ILLUMINATE_BULB_OKAY, REQUEST_SWITCH_SERVICE

FIGURE 8.2 State transition diagram (example)

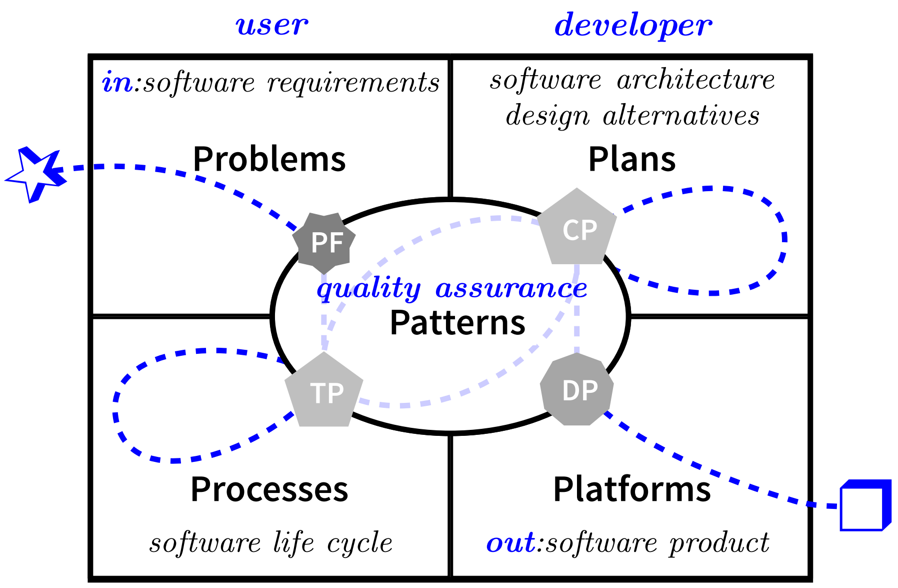
Business Process Model & Notation™

The Business Process Model and Notation™ (BPMN) [165] is an Object Management Group® (OMG) standard, which defines the notation and semantics for communicating process information. It provides for a modeling of business processes diagrams that bridge the gap between business user and technical developer. Business processes are in industrial practice the first class entity, i.e. the source from which software requirements emerge, and against which delivered value is evaluated.

8.3. All for One and "One4All" – An architectural view model for the three amigos

Figure 8.3 shows the "One4All" view model on software architecture, which is an adaption of Kruchten's "4+1 view model" as presented in section 8.2.1. Both models have in common that they

- account for the various perspectives and respective roles in a project team,
- establish correspondence among those different views,
- guide the project planning by "leading to a shared understanding for the team " [10].



Legend:

PF Problem Frames/Problem-Based FSM Patterns
CP Transition Templates (configuration pattern)
TP Transition Templates (task pattern)
DP Design Patterns, Frameworks & APIs

FIGURE 8.3 The "4+1" View Model becomes "One4All", adapted from [133, figure 1, page 43]

The "One4All" View Model applies the Three Amigos strategy common to agile software project planning. "The three amigos refer to the primary perspectives to examine an increment of work before, during, and after development" [10]. This strategy is attributed to Dinwiddie (2009), who found that these three people roles, i.e. the product owner or an analyst (*business user*), a programmer (*technical developer*) and a tester (*quality assurance*), "encourage at least three different [, mutually orthogonal] viewpoints [...] on almost anything you may be build" [80]. Dinwiddie resumes that, "Sometimes these three viewpoints are enough, but sometimes others are also needed. [...But] Looking at [these three perspectives] simultaneously and discussing tradeoffs from the different viewpoints provides further benefits, in terms of readiness [of development] and making better choices." [80]

In addition, the "One4All" View Model gives an answer to Dinwiddie, who states the question of "How could we record our decisions in a way that [...] other people[, i.e. all the members in a project team] will understand without repeating the entire discussion [of the amigos]?" [80]. He proposes the use of essential examples or acceptance scenarios, which “provide crispness to the understanding [. . .] that is hard to achieve any other way[. These serve to set up] a clear picture of the outcome before starting[, which keeps] the development on track.” [80]

Kruchten’s 4+1 View Model makes use of scenarios, which “ are in some sense an abstraction of the most important requirements [. . . and used] to show that the elements of the four views work together seamlessly” [133, page 47].

As figure 8.3 shows, the "One4All" View Model on software architecture enables decision-making and its documentation on the basis of *patterns*, which is – the One – view that glues in all the other views, i.e. *problems*, *processes*, *plans*, and *platforms*, for establishing problem-based project planning. *Patterns* guide the development of “blueprints” in each view and ensure correspondence among these. They help the team in avoiding different interpretations of the project plan, and in identifying “misunderstandings and confusions early and allows learning to happen sooner” [10].

The *problems*, and *processes* views are comparatively technology-agnostic. Accordingly, these belong to the user’s perspective, and are thus in the responsibility of respective roles in a project team, for instance the Product Owner, an Analyst, or a Domain Expert.

The *plans*, and *platforms* views deal with technology concerns, and thus belong to the developer’s perspective, who are familiar with the platform intented for building and operating the software product.

At the heart of the One4All View Model is the *patterns view*, which is responsible for arbitrating between the views of the user and the developer, and having an eye for developing (on) best practices[4]. It represents the quality assurance perspective in a project team, whose respective role may be taken by a Scrum Master, an Architect, an UI/UX Expert, a Tester, or a Project Manager, who is in general familiar with the problem as well as the solution domain. As this dissertation proposes an approach for mapping the patterns in each view, this provides options for tool support[5], which automates the responsibilities of the quality assurance role.

At last, The "One4all" View Model guides the pathway of the project team (see dashed line in figure 8.3) from the analysis of desired software functionality (starting in the *problems* view) to its delivery (ending in the *platforms* view). Thereby, the *patterns* view ensures a problem-based project planning, which not only makes a seamless transition between the different perspectives possible, but also helps the team in identifying and reason about those dependencies in requirements and architecture, which are too often unrecognized and thus accidently impede their work plans. The One4All View Model helps the team in identifying alternatives and precedencies for their project work plan, which increases their flexibility in deciding on what is reasonably to be done next.

The following sections explain the meaning and use of each view in detail.

[4]the team’s lessons learned

[5]for the rare case of affordable and available staff is not within reach

View 1. Problems

The "One4All" Problems view corresponds to the Logical view in the 4+1 View Model. Both views are comparably concerned with the partitioning of functional user requirements into more manageable units. Therefore, the Problems view collaborates with the Patterns view for establishing measurable units of software requirements. By means of tailored problem frames, namely problem-based functional size measurement patterns as have been developed in part II of this dissertation, the Problems view serves the scoping of desired software functionality and respective creation of Requirements Work Packages.

View 2. Processes

The "One4All" Processes view corresponds to the Process view in the 4+1 View Model. Both views have in common, that they serve the synchronization of software requirements. That is, they take the dependencies of requirements into account. This demands proper abstraction of desired software functionality “into separate threads of control that can be individually scheduled” [133, page 45]. For this purpose, the Processes view of the "One4All" view model is connected with the Patterns view, which provides respective patterns, i.e. problem templates to create basic units of activities, i.e. tasks (scenarios) for the requirements involved in a Requirements Work Package. The timely interaction of tasks across all Requirements Work Packages is then modeled by state transition diagrams, which represent the software life cycle.

View 3. Plans

The "One4All" Plans view corresponds to the Development view in the 4+1 View Model. Both views are in charge of mapping problems to solutions at the level of a software architecture design, one that envisions a "blueprint" of engineering work for satisfying the users expectations. The software architecture under consideration can refer to an actual design of a software already in place or one that is to be built. By reference to the Patterns view, which provides solution templates that ease the allocation of the problem model (requirements) to the solution model (configuration of architecturally significant computational components) on the same level of detail regarding their architecturally significant object(ive)s, the Plans view is capable of preparing several options for actions (candidate architectures or alternative solution designs) to build a coarse-grained design of a software system. These options represent units of (development) work, which enable the planning of how to fulfill a Requirements Work Package.

View 4. Platforms

The "One4All" Platforms view corresponds to the Physical view in the 4+1 View Model. Both views care in equal measures for the deployment of work units from development to the production environment, such that the software product becomes operative, i.e. released and its therewith delivered value is put into effect. By application of the One4All Patterns view, the platforms view takes advantage of design patterns, frameworks, and APIs to account for its responsibilities. This view is not further detailed in the following, but involved with the considerations given in the Future Prospect section.

View 5. Patterns

The "One4All" Patterns view corresponds to the Scenarios view in the 4+1 View Model. Both provide those abstractions necessary to glue the different views together, and act as a driver for a reproducible decision-making in the planning of expected team and development work; one which is based on software architecture design, cf. Kruchten [133, page 47]. The "One4All" Patterns view provides a collection of patterns for creating and reviewing software architecture design alternatives to the project team, and it defines essential relations among these patterns for ensuring consistency in the decision-making and use of best practices knowledge.

The following section Problem-Based Project Adaptation by the One4All View Model focuses on the application of problem templates, i.e. task patterns as designed in section 7.4, for establishing correspondence among the Processes view and the Problems view. This is needed to set up a software life cycle, which reflects functional dependencies among user requirements. Knowing these dependencies allows for reasoning about the useful order of delivering accordant software functionality, and adapt the project planning, respectively.

8.4. Problem-Based Project Adaptation by the One4All View Model

For organizing the planning of software projects Humphrey (1995) proposes the use of "a planning form. Now you don't need to decide what to do, the form tells you. All you do is fill in the blanks." [114, page 32]. The One4All View Model on software architectures makes this kind of forms in the Patterns view available. It is the heart of the One4All View Model, which offers templated structures namely patterns, that provide the team with those adaptation frames where change (to requirements) becomes controllable, and "flexing [of] what is being delivered" [21, page 42] becomes justifiable. The One4All View Model allows the team to adapt their project planning from multiple perspectives, each of these is supported by one view.

View	**Problems**	**Processes**	**Plans**	**Platforms**	The –One–4All **Patterns**
Concern	partitioning	synchronization	configuration	integration	**adaptation**
Container	RWP as the **problem model**, given as a set of tasks that share a common problem process	LCE as the **interaction model**, given as a set of temporally ordered tasks from several problem processes	ABP as the **solution model**, which is created by use of transition templates that make the problem absorb into platform	DO the **implementation** model, given as pattern-based description of the solution, which is scripted by the platform in use	·problem frames · transition templates · (architectural) design patterns
Component	domains (machine, environment)	problem process (control and data transitions)	(roles of) architectural elements (processors)	modules (framework)	
Connector	shared phenomena (causal, symbolic)	in and output events (trigger, data)	(forms of) architectural assembly (configuration)	interfaces (api)	
Notation	behavioral UML diagram (sequence diagram)	state transition diagram or LCE	transition schema	structural UML diagrams (class and component diagram)	depends on pattern

Legend:
RWP = Requirements Work Package, LCE = Software Life-Cycle Expression, ABP = Architectural Blueprint, DO = Deploy to operate

TABLE 8.1 Overview of adaptation techniques in the One4All View Model, cf. [133, URL table 1]

Table 8.1 gives an overview of **adaptation** techniques to the project planning, which are supported by and a specific concern of each perspective in the One4all View Model. The structure of table 8.1 is adapted from Kruchten [133, URL table 1].

The fifth, outer right column **Patterns View** allows for the scoping of problems and the bounding of requirements to a defined type and size of software functionality. Due to the introduction and use of patterns, requirements become representable at a common level of granularity. This is of importance for analyzing change and classifying its impact, e.g. the risk involved with it, systematically. For instance, cp. [21, page 42, section 6.4.3 Embracing change] change to an individual set of software functionality, e.g. modified details on one, independent problem, can be related with a minor change and addressed by other means, than change to (one problem in) a dependent set of software functionality, e.g. which can be seen as a major change, if crashing this dependence remains unrecognized and causes a business process to stop.

The first, outer left column **Problems View** serves the **partitioning** of requirements to measurable units of recognizable problems. It cares for a consistent separation and representation of these, such that in case of changing requirements, the scope of change is defineable and its size measured in function points changes correspondingly. This allows to choose proper risk mitigation, i.e. adaptation means, which drive the project planning. In case of the Problems View, it may become necessary to create additional problems for incorporating changing requirements, or to modify an exisiting problem by relating a new type or size with it. As a result, the project plan is updated. Part II of this dissertation elaborates the models relevant to the Problems View in detail.

The forth column **Platforms View** is not detailed further, following the assumption that from a user-centered approach to project planning, concerns and details of this perspective are either not of interest to the user, or coverable by the Plans View too. Nethertheless, the Platforms View is from a technical (requirements) perspective of special importance to the planning of software maintaince projects or reengineering activities. This aspect is discussed in the Future Prospects section of this dissertation.

The third column **Plans View** is about adapting the way or plan of how to solve a specific problem. Again patterns provide guidance for identifying those candidate architectures, which address the needs of each individual problem. Change within this perspective may result in adapting the selection of a solution candidate for one problem, e.g. by the “trading (or swapping)” [21, page 42] of one solution against an other. The choice of a different problem-solution-**configuration** represents an adaptation of the project planning. It is the structural dependence of requirements to architecture, which is addressed by the Plans View. The Plans View takes the framed problems, and enables the flexing of their proper solution. Part III in its chapter 7 elaborates the models relevant to the Plans View in detail.

The second column for the **Processes View** introduces an adaptation technique to the project plan, which considers the functional dependencies among the requirements, and thus determines and gives meaning to their (timely) precedence. The Processes View establishes a **synchronization** of the problems at hand. This dynamic perspective is represented as life-cycle expressions or state machines and elaborated in detailed next. This perspective creates an interaction model of requirements, which describes a "happy path" [200] of how desired units of software functionality work successfully together. The principle idea followed in the Processes View is to concatenate these units (user stories) from the Problems View, into a chain of basic activities, which represents a business process (feature or epic, cp. [21, page 222, section 25.4 requirements decomposition and granularity]). That way, equally formed and thereby scoped functional user requirements at the level of user stories serve as reusable and recognizable building blocks to plug value-delivering business epics together.

In order to execute the Process View, the requirements model from the Problems View must be available. Starting point are the task scenarios, which form a grouping of requirements into units of basic activities, whose type of functionality is known. Chapter 7.4 Problem templates explains in detail, how to set up these task scenarios. The Case Studies part gives examples for their application to a student recruitment and a vacation rentals management system. There, a discussion on respective software life-cycle expressions can be found, too. Section 8.6 presents an alternative requirements partitioning approach by use of UML use cases and how it fits with the One4all View Model. The next section 8.5 illustrates, how to set up the interaction model of the Processes View given as state transition diagram.

8.5. Synchronizing Requirements by a State Transition Diagram

After partitioning the requirements into basic units of activities (user stories) as is supported by the Problems View, these can be concatenated for modeling valid interactions between the user and the software under consideration (business epic) in the Process View of the One4all View Model.

Since each task scenario is built on patterns, for classifying the requirements type of functionality and for limiting their details to a recognizable level of granularity, each task scenario consists of

- one trigger, sent from a problem domain[6], that is received by the machine domain, and
- one corresponding action-event, which is triggered by the machine in response, intended for producing requested output,

[6] one, which resides outside the machine, such as an actor, or biddable, and causal domains

see chapter 7.4 Problem templates for more details.

Table 8.2 lists all task scenarios and their respective triggering- and action-event for the Student Recruitment Web Portal. Note: By chance, in this specific application example, there is exactly one requirement involved with one task scenario, and each requirements work package includes exactly one task scenario. As practices and the case studies part show, usually several requirements belong to one task scenario, and a requirements work package is made up of several task scenarios. This circumstance does not conflict with the following intent of illustrating the synchronization of requirements by composing task scenarios into a state transition diagram.

In addition, it is important to note, that for this purpose the focus is on the functionality and not on the data involved in some functional user requirements. That is, why parameter lists, etc. are not discussed at this point, even if they appear in some of the sequence diagrams in the requirements specification.

RWP	Task Scenario	see Fig./P.	IFPUG-EP/TOFF	trigger	action
FUR #02	Record Candidate Data	Fig. 11.13, P. 240	EI / TOFF-i.	record40FormData	store40FormData
FUR #05	Upload Candidate Files	Fig. 11.16, P. 242	EI / TOFF-i.	select6CandidateFiles	store6CandidateFiles
FUR #03	Review Candidate Data	Fig. 11.14, P. 241	EQ / TOFF-ii.	review40FormData	showCandidateData
FUR #06	Compile Candidate Résumé	Fig. 11.17, P. 242	EQ / TOFF-ii.	review40FormData6Files	showCandidateDataFiles
FUR #04	Download Candidate Data	Fig. 11.15, P. 241	EO / TOFF-iii.	review40FormDataToPDF	showCandidateDataToPDF
FUR #01	Grant Access Authorization	Fig. 11.12, P. 240	EO / TOFF-iii.	requestAccessAuthorization	sendURLviaEmail

TABLE 8.2 List of Task Scenarios for a Student Recruitment Web Portal

The level of detail created by use of patterns yields task scenarios, for which each can result only one defined effect (postcondition), one which is involved with one defined action-event triggered by the machine. That way, each task scenario can serve as a kind of state invariant or precondition for other task scenarios. This makes describing and concatenating these groups of desired software functionality at a constant level of granularity possible. It also avoids state explosion by artificially introducing state invariants, since it binds functional requirements dependencies to identifiable activities, and not to an arbitrarily placed labeling. A further noticeable fact about the property of task scenarios is, and which is shown by the example next, that those which represent an External Input (EI) elementary process or are of a corresponding type of functionality (TOFF-i), make the backbone of a process chain, which means

- 1st: their postcondition gives reason for a new state in the state transition diagram, and
- 2nd: task scenarios, which represent an EQ or EO (TOFF-ii., or TOFF-iii.) do not yield to the creation of new states in a state transition diagram, since these do not modify/write information to the system. This implies, that EQ/EO-task scenarios are usually attached to states involved with EI-task scenarios.

The benefit of these findings is, that the number of possible states in a state transition diagram can be limited considerably, and requirements dependencies can be maintained more effectively.

Figure 8.4 gives the state transition diagram for the Student Recruitment Web Portal.

In figure 8.4 the transitions triggered by a $candidate$ are colored in black, the ones that are triggered by the $admin$ are colored in gray. The triggers in **bold** text belong to a TOFF-i., i.e. it updates data hold by the software application.

The $trigger$ of each transition is marked as $< requirements\ work\ package > . < operation >$ and coincides with a respective message to the machine domain in one alt-fragment of a requirements work package. The $action$ taken when activating a transition is marked as $/ < operation >$ and coincides with a respective message to a constrained problem domain in the same alt-fragment

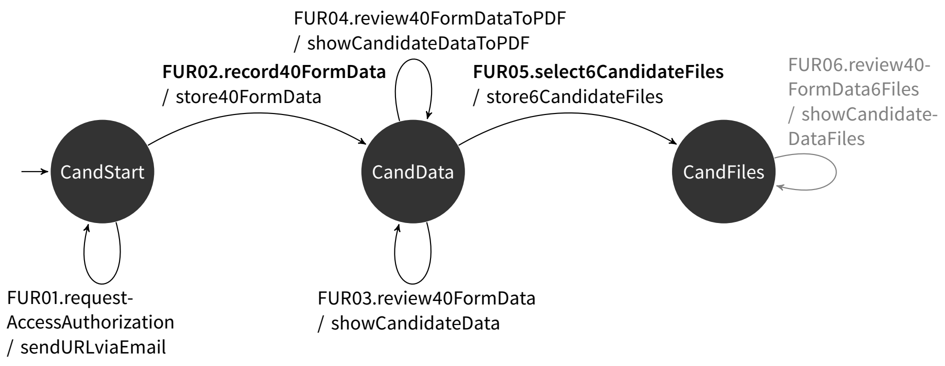


FIGURE 8.4 State machine as joint usage protocol for the student recruitment web portal

of a requirements work package. The transitions in figure 8.4 are consistent with the trigger- and action-column entires in table 8.2.

Now, the great benefit becomes obvious, of how patterns help to identify and model independent units of desired software functionality. These requirements units or task scenarios become synchronizable by the use of a state transition diagram. That is, their dependencies are representable and usable to manage their precedence (and importance) for a business process, which drives the project planning.

For instance, the critical path is along the task scenarios, which belong to a TOFF-i. (EI) problem. In the Student Recruitment Web Portal case, this means that a candidate must first submit personal data (trigger: FUR02.record40FormData), and then upload the application documents or respective files (trigger: FUR05.select6CandidateFiles) for completing the (workflow of the) application procedure successfully. The postcondition of FUR02 becomes a state (invariant) abbreviated “CandData” for the candidate data, that is now available for processing by the machine, and the postcondition of FUR05 becomes a state (invariant) named “CandFiles”, meaning that the candidates files upload is available for further processing too. The state labeling is optional. It should be expressive to the reader.

Task Scenarios of TOFF-ii. (EQ) or TOFF-iii. (EO) usually read information and do not write information to the software application. In a general case[7], a transition which represents these kinds of task scenarios, is attached to one state only. It starts and ends in the same state. It leaves the data information managed by the software unchanged. For instance, the task scenarios for FUR03 and FUR04 process and send information outside the machine, but the application procedure from a candidate perspective could do without these. Nethertheless, these two task scenarios depend on the availability of candidate data, which is established by (the postcondition of) FUR02.

The existence of states and the transitions between them in the Processes View is traceable to the functional user requirements that make up the Problems View, and maintained due to the use of the Patterns View in the One4all View Model.

[7]In an exceptional case, such as for FUR01: requestAccessAuthorization, which is classified as of TOFF-iii. (EO), it may be necessary to revise its classification, since its transition results in a new state “CandData”, as is usually only created by EI, or it should be checked, if this functional user requirements involves a yet unrecognized composition with an EI. For details on this, see the discussion section.

8.6. Sample Application to Use Case Decomposition

UML use cases [168] are a popular means for the grouping of functional user requirements. Due to its sketchy nature, the use case diagram is a visual gadget, which is a preferred one for quickly creating requirements documentations that (too often) result from communicating with the user (only). Use case decomposition blends well with problem-based requirements analysis as is shown next, with the result that the developer can also take advantage of this requirements documentation.

Tailoring use cases by task patterns (problem templates) into “basic units of activities” advances the use case descriptions and eases their consistent alignment with the software’s life cycle or business process model.

Part Case Studies demonstrates a respective use case decompostion for the vacation rentals.

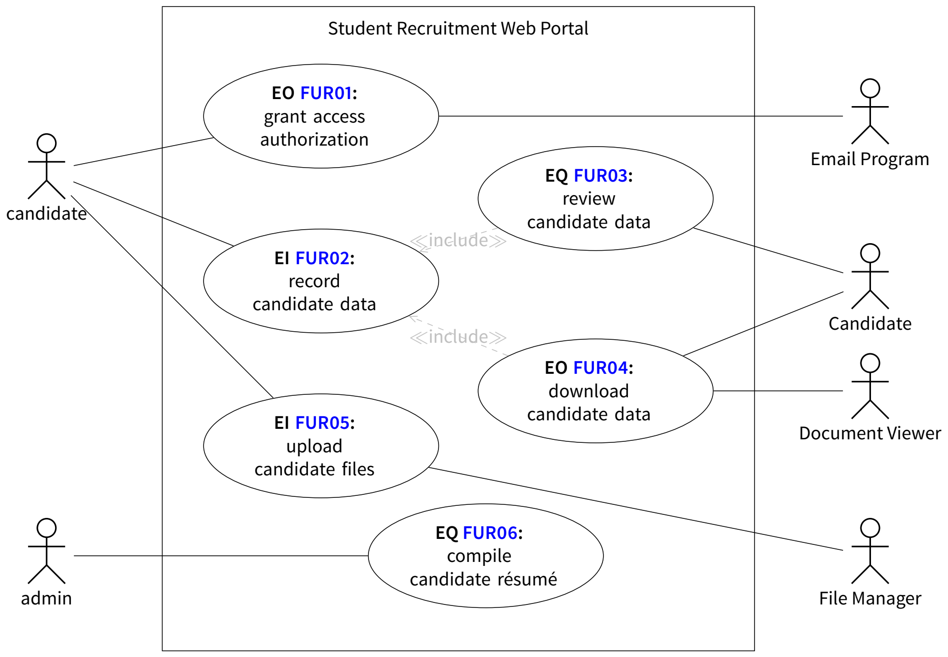

Student Recruitment Web Portal
EO FUR01: grant access authorization
EQ FUR03: review candidate data
EI FUR02: record candidate data
≪include≫
≪include≫
EO FUR04: download candidate data
EI FUR05: upload candidate files
EQ FUR06: compile candidate résumé
candidate
admin
Email Program
Candidate
Document Viewer
File Manager

FIGURE 8.5 UML use case diagram for the Student Recruitment Web Portal

Figure 8.5 gives a use case diagram for the student recruitment web portal, which takes advantage for the One4All view model. In this example, each of the six use cases (and their involved functional user requirements) is classified to one defined problem, which constitutes a specific type of functionality. This knowledge guides the creation of corresponding use cases, namely

- each EI or TOFF-i. use case must be connected with one triggering actor, and it has no participating actor, who is addressed by the actions invoked through the trigger.
- each EQ or TOFF-ii. use case must/should be connected with one triggering actor/include-use case, and it has no participating actor, who is addressed by the actions invoked through the trigger.
- each EO or TOFF-iii. use case must/should be connected with one triggering actor/include-use case, and it has exactly one participating actor, who is addressed by the actions invoked through the trigger.

In addition, each use can can be connected with further actors, which provide additional input to it. By following these validation conditions for a problem-based use case decomposition, the task patterns underling each use case are reflected in and thus yield a consistent use case diagram.

no.	use case	EP/TOFF	triggering actor	participating actor[8]	further actor[9]
E.1	FUR #06 compile candidate résumé	EQ/TOFF-ii.	admin	–	–
E.2	FUR #01 grand access authorization	EO/TOFF-iii.	candidate	email program	–
E.3	FUR #02 record candidate **data**	EI/TOFF-i.	candidate	–	–
E.4	FUR #05 upload candidate files	EI/TOFF-i.	candidate	–	file manager
E.5	FUR #03 review candidate **data**	EQ/TOFF-ii.	candidate by «include»	–	–
E.6	FUR #04 download candidate **data**	EO/TOFF-iii.	candidate by «include»	document viewer	–

TABLE 8.3 Reasoning on a problem-based use case decomposition

Table 8.3 summarizes the effects the validation conditions for a problem-based requirements partitioning have on the use case digram for the student recruitment web portal. It nicely illustrates the integration of desired software functionality (use cases) with the given environment (actors connected to each use case), which finally validates the application boundary or respectively the scope of the software system to-be, too.

The following explanations **E.1** to **E.6** discuss the entries in table 8.3 and their relevance for the use case diagram in figure 8.5. In general, a use case diagram gives an overview of independent actor-system-interactions. It does not define the timely order of its use cases. An «include»-relation means, that "The including UseCase may depend on the changes produced by executing the included UseCase. The included UseCase must be available for the behaviour of the including UseCase to be completely described" [168, page 641, section 18.1.3.3 Includes]. This kind of use case relationship reflects a functional dependence between requirements (and architecture), which is introduced in section 8.2.2 on page 157, and which becomes addressed and manageable by the One4all view model.

E.1 EQ-problems are concerned with compiling given information and simply presenting these on a display or other basic output device. In contrast to these are EO-problems, which compile information, and provide them (for control purposes via an external or definable interface) to other actors, software systems or programs, which do not belong to the software system under consideration (cf. **E.2**).

E.2 EO-problems differ from EQ-problems as they require a participating actor in the use case diagram (see also **E.6**). A participating actor is impacted by the actions/use case invoked by a triggering actor. In this example case, the student recruitment web portal makes use of an external interface to send an email containing an access link, which is received in an other software application, namely an email program.

E.3 The primary purpose of EI-problems is to receive some data from outside the application boundary and to make these persistent. The data filled in the forms by the candidate is simply recorded to make it available for further processing. The use cases FUR #03 and FUR #04 depend on these candidate data, which paves the way for considering this fact already in the use case diagram as elaborated for **E.5** and **E.6**.

E.4 EI-problems have no participating actor, but can have further actors, such as in this example the file manager program. It is another application than the student recruitment web

[9]participating actor = constrained causal problem domain
[9]further actor = referenced causal or biddable domain

portal. The file manager is to involve by the candidate for uploading selected documents to the storage of the software under consideration.

E.5 The use case for FUR #03 it triggered by the candidate, for reviewing the data already provided by invoking the EI-use case FUR #02, see **E.3**. Since the use case for FUR #03 represents an EQ-problem, which depends on the data stored via FUR #02, there is a structural dependence of these requirements to the architecture, which can be alternatively modeled via an «include»-relation in the use case diagram. The context remains the same, i.e. the candidate is still the triggering actor. In addition, it is already represented that there is a dependence on the common data information “candidate data”. If nothing is recorded, nothing can be reviewed on the screen. If the provided information is incomplete, so is the data compiled for the review. Note: this dependence does not force or mark a timely order of invoking these independent use cases for FUR #02 and FUR #03. Nethertheless, such a timely order can be defined with reference to this «include»-relation in the life-cycle expressions or state machines for the software life cycle or respective business process.

E.6 As for **E.5**, this use case for FUR #04 depends on the availability of “candidate data” recorded via executing the use case for FUR #02. The download of candidate data involves the usage of an external interface for exporting the compiled information into a pdf-file, which is accessable via another software application than the student recruitment system itself, namely a pdf- or document reader. This is one example, why EO-problems differ in their complexity from EQ-problems, which are equally concerned with processing data information, but do not involve other software applications, and rather make use of built-in display options.

The use case diagram in figure 8.5 gives an overview of the requirements partitioning (for the student recruitment web portal) into independent groups of basic actor-system-interactions (use cases at the granularity of user stories).

A state transition diagram can be used for modelling the software life cycle, i.e. the concatenation of user stories to business epics, one which indicates the order (precedence) in which units of desired software functionality should be reasonably delivered.

Figure 8.6 gives a state machine diagram, which builds on the use case diagram in figure 8.5 for modeling the software life cycle of the student recruitment web portal.

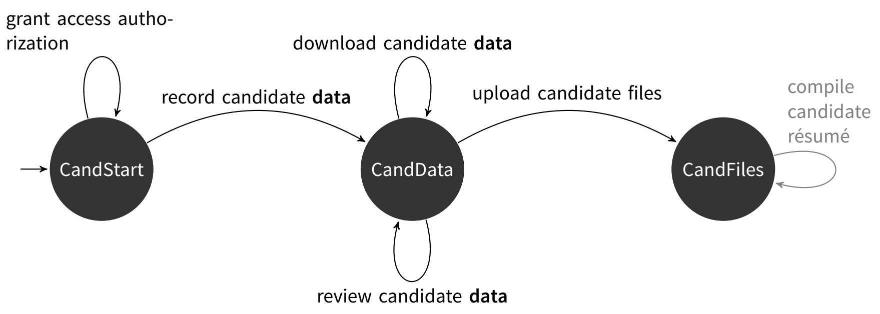


FIGURE 8.6 State machine by means of UML use cases for the Student Recruitment Web Portal

By application of a problem-based requirements partitioning for modeling use cases, these form equally scoped units of self-contained software functionality. Thereby, each use case becomes by its execution a provider of a unique and identifiable postcondition, that can be of relevance as precondition to an other use case. That way, recognition of functional requirements dependence is made easier and better usable for the project planning.

8.7. Discussion & Related Work

Alebrahim (2017) proposes in [3, 4] a method for problem-based detection of functional requirements interactions, which addresses the case where the satisfaction of one requirement affects the satisfaction of another [185], and how this situation can be relaxed. In this context, interactions are seen as particular correlations between requirements, e.g. requirement dependencies, which have according to Robinson et al. [185, table 3, page 18] none, undefined, negative or positive consequences on their satisfaction.

In order to start Alebrahim's method, a comprehensive list of functional software requirements, and a software life cycle model, which determines the precedence of the requirements, must be available beforehand as external input to this method [3, page 201]. Then, the requirements list is analyzed in a three-phase pruning process for identifying interaction candidates. Therefore, assumed and effected environmental conditions (system states) as are documented by the requirements specification and applicable in *phase 3: precondition-based pruning* and *phase 1: structure- or postcondition-based pruning*, as well as the precedence of these conditions as applied in *phase 2: life cycle-based pruning* are taken into account.

Problem-based requirements interaction analysis and the One4All View Model build in equal measures on "Shared domains [to analyse requirements dependencies, since these] provide points in the requirements model where requirements [can] interact." [3, page 196]. This property makes the link between these two approaches. Due to its use of requirements work packages, which maintain requirement dependencies at the level of problems, as well as the application of a software life cycle model, that maintains requirement dependencies at the level of tasks, the One4All View Model improves the input to and thus the quality of problem-based (interaction) analysis for functional requirements, since reasoning on these and thus trade off considerations become much more straightforward and reproducible. For instance, problem diagrams combined to requirements work packages ease structure-/postcondition-based pruning, since all the requirements in a work package address the same constrained problem domain. Furthermore, the software life-cycle model, which operates on the structuring of requirements into tasks as in the One4All View Model, eases life cycle-based as well as precondition-based pruning, since the thereby defined level of granularity for tasks assures managing traceability of and the dependency among requirements.

Choppy et al. [51, 52] propose the use of system states for composing a state machine, which expresses the software life cycle. By means of UML state invariants, which represent the pre- and postconditions of problem-based requirements specifications, the logical order of problems and therewith involved satisfaction of the software's requirements is defined. The One4All View Model extends this approach by providing means for limiting the risk of state explosion, which is demonstrated in part V Case Studies in more depth. This is achieved by considering task scenarios, which form a requirements specification by basic activities only. In consequence, each task has only one trigger, which can cause a transition in the state machine. Furthermore, new state invariants or system states do not rely on a modeler's experience anymore, and maybe accidently introduced. Systems states with relevance for the overall software life cycle are bound to problems that process received information, i.e. problems that belong to TOFF-i. (external input) software functionality as introduced in II Problem-Based Project Estimating. That way, other kind of problems, such as of TOFF-ii. or TOFF-iii. can be reasonably ignored when introducing new system states. As a result, the One4All View Model advances the precision by which the composing a software life cycle and its extension in case of emerging changes to the requirements is made feasible.

8.8. Summary

This chapter introduces a comprehensive architectural framework called the "One4All View Model".

It joins the different views present to a project team based on pattern practices. That way, the various perceptions of the work required to be done for delivering desired software functionality are brought into one coherent perspective. This eases the team's decision-making on how to proceed and to agree a corresponding project plan.

Therefore, the One4All View Model represents an adaptation framework as detailed in section 8.3, which enables problem-based project planning. As introduced in this chapter, this problem-based adaptation framework makes explicit allowance for the functional as well as the structural dependencies of software requirements, see page 156ff. It advances the resolution of requirements interactions by taking the architecture and the software life-cycle into systematic account.

In case the satisfaction of one requirement affects the satisfaction of another [185, page 6], the Plans View and the Process View of the One4all View Model helps the team to identify and analyse this circumtance. The Plans View makes use of transition templates as introduced in the previous chapter 7, for looking at the shared computational components in an architecture design on which the requirements depend. The Processes View considers the software life-cycle as a means for requirements synchronization. It models the functional precedence of desired software functionality, for satisfying (business processes) and thereby in which order best to deliver these. The use of state transition diagrams is elaborated in section 8.5 of this chapter for representing this dependency relation between (equally scoped units of) software requirements. Both, architecture and software life-cycle build on requirements at the level of tasks, which are created by the Patterns View in the One4All View Model.

Having these tools for requirements dependency analysis at hand, the team is in the position to check and adapt their project planning from an engineering as well as a management perspective. It is the utilization of patterns in each perspective, which enables the team to decide on a defined and mutually understandable basis.

They can explore alternative solutions by different architectural configurations or they can benefit from the requirements precedence given in the software life-cycle for their project planning. In addition, these tools ease determining the criticality (minor or major change) of the software requirements, and conflicts become resolvable by trading these instead to cut-off, i.e. to prune desired software functionality.

Part IV.

Problem-Based Project Benchmarking

*Part IV Problem-Based Project Benchmarking is about answering **RQ 1** How to compare speed? It integrates Problem-Based Project Estimating and Problem-Based Project Adaptation to an agile project process framework introduced as A S.M.A.R.T. Scrum-A·Gen^{E}DA for taking advantage of a project plan (project backlog) that builds on point values (product size) as measurement for a defined set of software requirements (product scope). This kind of project plan serves as baseline for establishing benchmarks (project speed), which are comparable among projects and teams. Chapter 9 Problem-Based Project Baseline and Speed Benchmark details the use of Requirements Work Packages and the "One4All" View Model on software architecture as developed in this dissertation to set up units for measuring project work progress. Applying these for establishing a work plan of software product requirements and its involved performance baseline for a project, ensures the comparability of speed benchmarks. Benchmarking a Problem-Based Project Baseline – A sustainable planning game demonstrates the use of these units for empowering software project teams to benchmark their project success (points scored) compared to their project plan (points committed), whenever a project timebox is completed. Conducting agile projects by A S.M.A.R.T. Scrum-A·Gen^{E}DA implements problem-based project planning, which comes with built-in means for requirements prioritization and for exploring alternative solution designs. In addition, it supports software project teams in adjusting their decision making and development activities as needed, both proactive and retrospective. Ultimately, it makes sustainable control of software projects possible, for and by a demonstrable delivery of value.*

9. Problem-Based Project Baseline and Speed Benchmark

9.1. Introduction

As Meyer (2014) comments, “Successful project control requires both *estimation* of effort, in advance of an iteration, and measurement of progress, during the iteration and at the end. [...] For both estimation and *measurement*, teams need units of progress.” [156, page 121] These units of progress are introduced in this dissertation by Requirements Work Packages, which provide for reproducibly scoping and sizing of desired software functionality. By use of Requirements Work Packages not only (functional size) estimation in advance of a project (iteration) is possible, but also measurement of progress in terms of successfully "done" these units is supported. Thus, several benchmarks become available: starting with the (function) points (as size) estimated for the items that belong to the project baseline, to which the team commits to in the beginning of a project, followed by the points scored in between for "done" work items, i.e. Requirements Work Packages, and finally the total of points scored for delivered software functionality, after the project time-box is completed. It is the Project Backlog, which serves as this project baseline providing for “a reference level against which an entity is monitored and controlled” [21, page 325], and against which to benchmark project success. In order to illustrate where and how the findings of this work advance the comparison of project performances (speed), these are integrated to the most commonly used agile project process framework named Scrum in the following.

Section 9.2 Background gives a brief introduction to Benchmarking and Scrum.

Section 9.3 Make the Frame(s)work rebuilds the Scrum framework by means of the Agenda Concept to setup problem-based project planning as described by A S.M.A.R.T. Scrum-A·Gen^{E}DA[1]. For each Scrum Event of Planning, Daily, Review and Retrospective a corresponding Agenda is given, such that this agile project process can be executed in a problem-based way. Respectively, the Scrum conceptions of Team and Artifacts are developed further to integrate well with the ones newly introduced in this dissertation, namely with patterns as applied in the "One4All" View Model of software architectures.

Section 9.4 Benchmarking a Problem-Based Project Baseline – A sustainable planning game illustrates how problem-based benchmarking sustains projects and teams. It gives a sample application taken from the case studies.

Section 9.5 Discussion & Related Work looks at the use of benchmarking for the continuous improvement of projects and teams.

Section 9.6 Summary gives the benefits for agile project planning when relying on a problem-based project baseline and speed benchmark as provided by A S.M.A.R.T. Scrum-A·Gen^{E}DA.

[1]This it is a word play, which refers the research project GenEDA [109], the Agenda Concept [106], the writing of S.M.A.R.T objectives [82], and Scrum [193] as combined in here.

9.2. Background

This section gives the state-of-the-art literature and concepts used in the following for developing A S.M.A.R.T. Scrum-A·Gen^{E}DA, which is a benchmarking approach to problem-based project plans. It allows for their comparison in regard to size-driven scope and speed considerations.

Section 9.2.1 Benchmarking introduces the need to take advantage from the lessons learned in each project, such that 'continuous improvement' can happen.

Section 9.2.2 Scrum picks the concepts out of the Scrum framework, which are used in the following to set up A S.M.A.R.T. Scrum-A·Gen^{E}DA.

9.2.1. Benchmarking

Camp's work published in 1989 on the topic of *benchmarking* is considered a cornerstone in this field. To him "Benchmarking is a positive, proactive process to change operations[, e.g. on the software delivery process] in a structured fashion to achieve superior performance." [48, page 3] This is achieved only by "the comparison to and the understanding of the best practices [...of others that] will ensure superiority." [48, page 4]

In this context, performance is not considered in isolation, such as looking only at the measurable output of a process, e.g. the speed of delivery. It also incorporates the understanding and implementation of those practices, which result performance improvements. It is true that "Benchmarking implies measurement." [48, page 11] and "The most significant metric for software projects is "size"." [194, line 26], but benchmarking is not only about performance measurement, it is also about "enhanced performance by learning" [113, page 1].

As "truly innovative ideas are probably more likely to be found [...] outside one's own industry." [113, page 6], 'creative swiping' [174] is a means for "Emulate their strengths." [48, page 4] This search and *adapt for the better* is characteristic for continuous improvement processes and leads to the realization that "Benchmarking is entirely consistent with 'kaizen' (Imai 1986) ." [113, page 3] Therefore, benchmarking "forces constant testing of internal actions against external standards of industry practices. [And] It promotes teamwork [...] to remain competitive." [48, page 15] and to focus on incorporating "those practices into their [own] operations [...that] meet customer needs and have a competitive advantage. " [48, page 3] That way, "ensuring the organization is satisfying customer requirements [by their delivered products] and will continue to do so as customer requirements change over time." [48, page 21] "The purpose of benchmarking is to ensure that probability of success." [48, page 3]

"Benchmarking is an integral part of the planning [...] to strengthen the use of factual information in developing plans." [113, page 2] "It assists managers in identifying practices that can adapted to build winning, credible, defensible plans." [48, page 15] Benchmarking provides valuable input to the project planning and guidance in the project execution [113, page 23]. By its strict reference to proven problem-solving know how (best practices), it improves[2] the planning of a project

- in the beginning, as it ensures more realistic commitments that win the user, e.g. can the problem be solved on schedule?
- during project execution, as it serves for a more flexible and defensible decision making, e.g. how to solve the problem effectively? Are we on track?
- at the end, it establishes credible knowledge on the efficiency of what works and what does not, i.e. has the plan worked out, such that the way of solving a problems is a worthwhile one, e.g. as a lessons learned for future plans?

[2]and used as a means for risk management it can also demonstrably disprove a plan

9.2.2. Scrum

Scrum is an agile project process framework developed by Schwaber and Sutherland, of which the latest version is published in 2017 as a 20-pages document named The Scrum Guide™ [193]. It describes the roles, responsibilities, and resources of an agile project organization, and summarizes them into rules of the game to enable a highly-frequent and adaptive problem-solving process that promotes collaborative work and value-based implementation. This type of project implementation has become particularly popular in the area of software development. Since Scrum in itself is neither a method, nor comes with specific techniques or technologies for its execution, an adoption of Scrum is developed in this chapter. Therefore, the following section 9.3 Make the Frame(s)work is structured in exactly the same way as The Scrum Guide™.

9.3. Make the Frame(s)work

This section elaborates, how function points not only serve the comparability of software products, but also advance the comparability of software production performances by size-driven speed benchmarks.

9.3.1. Project Time-Box

Table 9.1 shows that Iteration, Cycle, or Sprint are various terms that are used analoguously for describing the time frame available for starting, managing, and closing a project process with which a desired project result is to be achieved, e.g. a potentially releasable increment of software product functionality.

Estimate&Measure [156]	in the beginning	during	at the end	Time-Bound
		Project Iteration		
to compare **speed** for	capacity	progress	success	**benchmarks**
PDCA [76]	Plan	Do	Check&Act	Steps
		Continuous Improvement Cycle		
Scrum [193]	Planning	Daily	Review, Retrospective	Events
		Sprint		
A S.M.A.R.T. Scrum-A·Gen^{E}DA [sec. 9.3.4]	Planning	Daily	Review, Retro	Agendas
		Project Time-Box		

TABLE 9.1 Benchmarking Periods within the Project Process

Within an agile project setting, this project time-box for executing the project process is fixed. Effective software project control, which in agile means to continuously "inspect&adapt", requires estimation and measurement in the beginning, during, and at the end of such a project time-box. Comparable speed benchmarks are useful in many ways. In the beginning of a project time-box, a (credible) speed benchmark helps to set up a reasonable plan of the work volume or capacity a team is capable to achieve within the time available. Speed benchmarks available during the project time-box indicate the progress made up to now. In case of deviation from plan, respective countermeasures can be taken. Speed benchmarks at the end of a completed project time-box show the success achieved by the team to deliver desired project results in time. The ratio of remaining work in progress and the work successfully "done" indicates the credibility of a project plan. Project teams can use this benchmark in future project planning for justifying the work volume they can usefully commit to. It is a performance measure, which represents the team's "sustainable pace", one at which it has demonstrated to be capable of achieving desired project results.

Agile project processes in general implement a continuous improvement cycle, to which the four Scrum Events of Planning, Daily, Review and Retrospective can be mapped respectively. Both, Review and Retrospective occur at the end of a project time-box, but each serves a different purpose. These differences are explained in the Agendas of A S.M.A.R.T. Scrum-A·Gen^{E}DA, which are structured into the project time-box comparably to the Scrum Events.

9.3.2. Project Team

A Scrum Team consists of a Product Owner (PO), a Development Team (SE, QA), and a Scrum Master. There is no role like a Project Manager, since in agile project teams are self-organizing, and the responsibility for delivering desired project results is shared among all Scrum team members.

Adapting a project process to an organization involved always the mapping of the process roles to the members in a project organization. Since problem-based project adaptation as introduced in section 8.3 makes use of the Three Amigos, these build the Project Team members in the Scrum-alike A S.M.A.R.T. Scrum-A·Gen^{E}DA.

The role of the Scrum Master is substituted by the Product Owner in the following. This may be criticized as not being Scrum anymore, but it still implements the Scrum framework to an extent, which suffices for implementing the needs of agile project management.

PO

The Product Owner role is responsible for managing the Product Backlog, clearly expressing the Product Backlog items, and prioritizing these to optimize the value of the work to be “done”. The Product Backlog is the only source of requirements, the team is allowed to work from.

The Product Owner as defined here serves the other project team members as a Scrum Master. The PO cares for the team by ensuring a common understanding of what and how work is to be done, and by leading and coaching the Project Team, or removing impediments respectively.

QA and SE

The Quality Assurance or Tester role and the Software Engineer(ing) or Developer role are individual Project Team members, who have specialized skills and areas of focus [193, page 7].

The Engineer is mainly concerned with proposing and implementing ways to get the work done. QA defines and checks respective acceptance criteria. In this context, QA cares for proper integration of project results to the evolving software product (by regression testing, and conformance check with any organizational standards) and (by lessons learned) to the improvement of the project process and the team's maturity.

QA and SE work collaboratively with the PO to deliver a “done” product increment. Consequently, all team “members must have a shared understanding of what it means for work to be complete” [193, page 18]. Furthermore, the same definition of “done” guides the Project Team “in knowing how many Product Backlog items it can select during a Sprint Planning.” [193, page 18]. So, problem-based project benchmarking as established by A S.M.A.R.T. Scrum-A·Gen^{E}DA ensures exactly this definition of "done".

9.3.3. Size Matters to Keep the Pace

As function points are used in Problem-Based Project Estimating to determine the size of a software product with respect to its functional user requirements, the use of these point values is continued in problem-based project benchmarking to be of equal advantage for determining the speed of the software production process, which is usually referred to as the pace or performance, within which a project team has demonstrated their capability to deliver at deadline a “done” product increment.

Product Increment

Agile software development conforms to evolutionary software production. Either a software product already exists, which is to be developed further in an incremental way, or there is no product increment yet, but an initial set of desired software functionality in the Product Backlog, which result from models, wireframes, prototypes, or legacy systems to be replaced. A product increment is a body of inspectable, done work at the end of a Project Time-Box, which sums all the Product Backlog items completed [193, page 17].

Product Backlog

A decision on what requirements should be done first in the upcoming Project Time-Box is driven by the items’ ordering in the Product Backlog. Higher ordered items are usually more refined, added with details, estimates and priority, such that these are more likely "ready" for selection in a Project Time-Box. The Product Backlog contains items "done", which are already implemented in the product increment, and lists those items to be done for making the product increment complete. “It is the single source of requirements for any changes to be made to the product” [193, page 15].

Project Backlog

The Project Backlog is named Sprint Backlog in Scrum. It lists all the estimated Product Backlog items selected by the Team to be done in a Project Time-Box, together with a work plan of how the team intends to achieve desired project results.

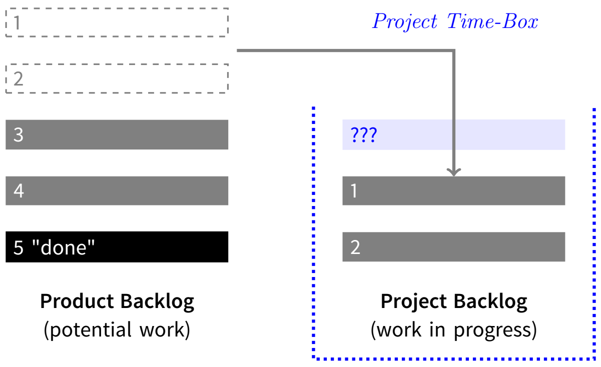


FIGURE 9.1 Product Backlog, Project Backlog, and the time available, adapted from [180]

The dotted line in Figure 9.1 marks the Project Time-Box available for a Project Backlog. Its capacity is the maximum, reasonable number or size of selectable Product Backlog items for a Project Time-Box, which should be aligned to past performances, i.e. the speed of the Project Team. It must be known to the team and its project planning, how much work (items) fit into the Project Time-Box available.

9.3.4. A S.M.A.R.T. Scrum-A·Gen^{E}DA

Problem-based project benchmarking requires estimating and measuring of project work items at defined periods within the project process. In addition, these must form comparable units of progress, so that the Project Team is in the position to judge its performance.

As motivated in section Size-Driven Software Project Planning, these units of progress applicable in a project plan, and which are meaningful in the beginning, during, and at the end of a Project Time-Box, should be S.M.A.R.T. [82], so that they serve the different perspectives, i.e. Management and Engineering, present in a Project Team equally.

As illustrated in table 3.2 on page 21 and elaborated throughout this dissertation, Requirements Work Packages, which are based on and take advantage of patterns, fit this need. They are applicable as project work items that guide the definition of

- **S.pecific** objectives or project results by packaging sets of desired software functionality
- **M.easurable** project work items by determining a function point value for these
- **A.chievable** and alternative solutions as action plans to produce desired project results
- **R.ealistic** commitments, as past performance justifies consensus about future work plans
- **T.ime-bound** risk assessment and delivery forecast based on "done" product increments

size-data useful to baseline and benchmark the software product as well as the software production process.

The following sections integrate the concepts developed in problem-based project estimating and problem-based project adaptation to an agile project process framework, which gives an account of Scrum in form of Agendas, for establishing problem-based project benchmarking.

The resulting A S.M.A.R.T. Scrum-A·Gen^{E}DA implements Problem-Based Project Planning, which enables the project team to control their speed of project success by patterns and function points.

Project Planning – Baseline product scope

The Project Planning Agenda in table 9.2 represents the Scrum Event of Sprint Planning applicable to problem-based project planning.

Name:	Project Planning
Input:	· Product Increment, latest (if one already exists) · Project Time-Box ::= {duration of 14 or 28 days}, available · Past Performance of Team ::= {scored points per Project Time-Box} · Projected Capacity of Team during Project Time-Box :: = {points per Project Time-Box} · Product Backlog ::= {ordered list of requirements[0]}
Participants:	Team ::= {3Amigos, stakeholders[1]}
Activities:	P 1. Craft a Project Goal. P 2. Estimate and decide on Product Backlog items for the Project Time-Box. ▶ *Planning Poker or* ▶ ***Problem-Based Project Estimating***[2] P 3. Plan how to deliver Product Backlog items into a "done" Increment. ▶ ***Problem-Based Project Adaptation***[3]
Output:	· Project Goal · Project Backlog ::= {set of estimated Product Backlog items selected for the Project Time-Box, plus their work plan for converting them into a Product Increment} as **Baseline**
Validation:	· Each Project Backlog item is assigned with a point value. · Each Project Backlog item is assigned with a (definition of "done" for its) work plan.

[0] requirements packed into Requirements Work Packages
[1] stakeholders to provide technical or domain advice
[2] Frame Counting Agenda
[3] Plans View of the "One4All" View Model on software architectures

TABLE 9.2 Project Planning Agenda

In the beginning of the Project Time-Box is the project planning, which serves the establishment of a problem-based project baseline. It contains all the work items from the Product Backlog, the team has selected to be subject to their upcoming work plan. The Project Backlog represents this baseline. It is a set of estimated requirements the team has committed to be "done" after the Project Time-Box is completed.

The Project Backlog is the resulting output from executing all Project Planning Agenda activities. In addition, all validation conditions should be checked, before continue working on the Project Backlog by the subsequent Project Daily Agenda. The validation conditions ensure the quality of the Agenda output.

In addition to the three amigos that make up the Project Team in problem-based project planning, stakeholders from business and IT can participate in this meeting to contribute their insights to the items of the Product Backlog. In problem-based project planning the Product Backlog is an ordered list of Requirements Work Packages applied as backlog items.

The Product Increment can be available in various conditions. It provides a point of reference to the team for decision making on their project work plan.

Past performance of the team is (the speed) measurement resulting from problem-based project benchmarking, which serves as input to project planning. It can be the case, that this point value (in an exploratory, zero-project time-box) is not yet available. The projected capacity of the team depends on this performance measure. For instance, if a team can usually do 10 points in a Project

Time-Box, it cannot reasonably commit to achieve this performance in case of absent Team members on holiday leaves. After all, the Project Time-Box must have a fixed duration so that the performance measurements are comparable.

The activities section in the Project Planning Agenda contains three steps for project planning. First, *Craft a Project Goal*, should serve the team as an overall theme for the work, they plan to accomplish in one Project Time-Box.

Second, *Estimate Product Backlog items in order to decide which to include in the Project Time-Box*, is "traditionally" done in agile by Planning Poker. Problem-based Project Planning makes use of ▶ Problem-Based Project Estimating for executing this activity. Starting from the top of the Product Backlog, a function point value is determined for each backlog item that is present as Requirements Work Package. This product size measurement (on the basis of functional user requirements) becomes a project performance or speed measurement in problem-based project benchmarking. This second activity coincides with the patterns used in the Problems View of the "One4All" View Model of software architectures. That means, the size determined for the problem to be solved depends on the pattern used to classify respective requirements too.

Third, *Plan how to deliver Product Backlog items into a "done" Product Increment* is executable by ▶ Problem-Based Project Adaptation as proposed in this dissertation. The Plans View provides solution candidates for each Requirements Work Package on the basis of patterns. That way, each Project Backlog item can be assigned with a(n alternative) work plan or architectural blueprint of how to accomplish desired project results.

With the help of the Project Planning Agenda, the Project Team can establish a project baseline of defined product scope that is promisingly "ready" to be successfully carried out in the Project Time-Box. This plan of project work to be done is measurable and analyzable in terms of successful and failure performance, and is particularly useful to reduce the risk of incorrect assumptions and involved false commitments.

Project Daily – Work In Progress

The Project Daily Agenda in table 9.3 represents the Scrum Event of Sprint Daily applicable to problem-based project planning.

Name:	Project Daily
Input:	· Project Goal · Product Increment, latest · Project Backlog ::= {set of estimated Product Backlog items plus their work plan} · Project Time-Box ::= {remaining days}, for Ongoing Project Backlog
Participants:	Team ::= {3Amigos}
Activities:	D 1. Do work on Project Backlog items. D 2. Inspect and adapt for project's work progress.
Output:	· Product Increment ::= {working set of “done” software functionality} · Project Backlog, items “done” are marked respectively · Ongoing Performance of Team ::= {scored points for “done” Project Backlog items}
Validation:	· Work on Project Backlog items is still aligned with Project Goal and Time-Box. · Project Backlog items "done" are marked and respective point values score to the Ongoing Performance of the Team.

TABLE 9.3 Project Daily Agenda

The items of the Project Backlog agreed by use of the Project Planning Agenda represent the work in progress for a Project Time-Box. During the daily project work on these items, they become "done". Therefore, the Project Team realizes the items' assigned work plan, and delivers these as a working, potentially ship- or releasable Product Increment. Either routine, day-to-day business, or creative project work, each comes with the risk of operational surprises (impediments), which hinder the team's progress as planned. That is why the Sprint Daily in Scrum “is a key inspect and adapt meeting” [193, page 12] carried out in a Just-In-Time Fashion, daily during an Ongoing Project Time-Box. As Scrum's Sprint Review and Sprint Retrospective, which both take place at the end of a Project Time-Box, serve the same purpose of "inspect&adapt", this issue is considered by the respective Agendas in more detail.

Of relevance in the Project Daily Agenda to problem-based project benchmarking is that each work item "done" is marked in the Project Backlog, and that respective point values score to the Ongoing Performance of the Project Team.

The ratio of work successfully "done" and the work (still) in progress is often represented in a Burndown Chart to illustrate the project's progress. In case of deviation from plan, problem-based project adaption can be considered for identifying alternative options for action and thus for delivering desired software functionality in the remaining time.

Project Review – Benchmark project success

The Project Review Agenda in table 9.4 represents the Scrum Event of Sprint Review applicable to problem-based project planning.

Name:	Project Review
Input:	· Product Increment, latest · Project Backlog, of Ongoing Project Time-Box · Project Time-Box ::= {all days exhausted }, time for Ongoing Project Backlog is elapsed · Product Backlog, as before Ongoing Project Time-Box
Participants:	Team ::= {3Amigos, key stakeholders}
Activities:	R 1. Inspect Product Increment. R 2. Adapt Product Backlog. ▶ ***Problem-Based Project Adaptation***[1]
Output:	· Product Backlog, revised (in priority and presence of items) · Ongoing Performance of Team ::= {scored points for "done" Project Backlog items} as **Benchmark**
Validation:	· All Project Backlog items can be found and are up-to-date in the Product Backlog. · Those Project Backlog items not "done" are considered (for their priority and presence) in the revision of the Product Backlog. · Ongoing Performance of Team represents the **speed** at which it has successfully delivered desired software functionality.

[1] Processes View of the "One4All" View Model on software architecture

TABLE 9.4 Project Review Agenda

At the end of the Project Time-Box the Project Review Agenda and the Project Retro Agenda are executed together to benchmark the overall project success. The project review serves the closure of the Project Backlog and preparation of the Product Backlog for the next Project Planning. It focuses the product and its (not yet) satisfied requirements compared to the baselined scope in the beginning of the Project Time-Box. The project retro serves the continuous learning of the team. It focuses the production process and related (past) project performance compared not only to the baseline plan in the beginning of the Project Time-Box.

The Project Review Agenda consists of two activities.

First, the Project Team together with the key stakeholders *inspect the "done" Product Increment* at the end of the Project Time-Box, where “the presentation of the Increment is intended to elicit feedback and foster collaboration [...] on what to do next” [193, page 13]. Since at this time, all days of the Project Time-Box are exhausted, the Ongoing Performance of the Team is now fixed. That is, the speed at which the team is capable to deliver desired project results per Project Time-Box is now known. It is a benchmark of a now available "done" Product Increment against the planned product scope in the beginning of the Project Time-Box and documented in function points as related with respective Project Backlog items.

Then, the *Product Backlog is adapted*, which involves an update of its work items and their status. Some work items have been "done", others are started but remain work in progress, still others did not change. Furthermore, during the daily work on the Project Backlog, the need for new work items may have been identified by the Project Team, or some requirements may have become obsolete.

By ▶ Problem-Based Project Adaptation, these changes to the Product Backlog become understandable. Here, the Processes View of the "One4All" View Model is helpful, since it allows for incoporating desired changes to the Product Backlog, taking requirements dependencies into account, which are represented as the software product’s life-cycle (business workflow). It supports the team in deciding on the priority of an item in the Product Backlog and its general presence in the "done" Product Increment.

The output of the Project Review is a revised Product Backlog and a performance measurement, which indicates the ongoing pace of the project team and enables a more realistic planning of subsequent Project Time-Boxes in regard to their capacity.

Project Retro – Lessons Learned

The Project Retro Agenda in table 9.5 represents the Scrum Event of Sprint Retrospective applicable to problem-based project planning.

Name:	Project Retro
Input:	· Product Increment, latest · Project Backlog, of Ongoing Project Time-Box · Project Time-Box, used for Ongoing Project Backlog · Ongoing Performance of Team ::= {scored points for "done" Project Backlog items } · Past Performance, of other Projects or Teams · Product Backlog, revised
Participants:	Team ::= {3Amigos}
Activities:	R 3. Inspect Team Performance. ▶ ***Problem-Based Project Benchmarking*** R 4. Adapt for Team Improvement. ▶ ***Problem-Based Project Adaptation***[1]
Output:	· Project Backlog, enshrines Lessons Learned, which are imported to the Product Backlog *by* ▶ ***recognizable units of proven problem solving knowledge***[1] as a shareable means to improve Team work/performance in next Project Time-Box. · Project Time-Box, revised capacity
Validation:	· Cause of Ongoing performance has been identified, and considered in Lessons Learned. · Ongoing Performance of Team is taken into account as past speed for decisions on the teams projected capacity (sustainable pace) in subsequent Project Planning Agenda.

[1] Patterns View of the "One4All" View Model on software architecture

TABLE 9.5 Project Retro Agenda

The last but not least Agenda to be executed at the end of a Project Time-Box is Project Retro. It is only the Project Team, which participates in this meeting, since it serves their continuous improvement by holding on their lessons learned.

No changes to the product's requirements benchmark, i.e. the "done" product increment, or to the project's performance benchmark, i.e. the team's scored function points for this Project Time-Box, are made, but these benchmarks are compared with the baseline given by the Project Backlog of the Ongoing Project Time-Box, or others if available.

The Project Retro is organized into two activities.

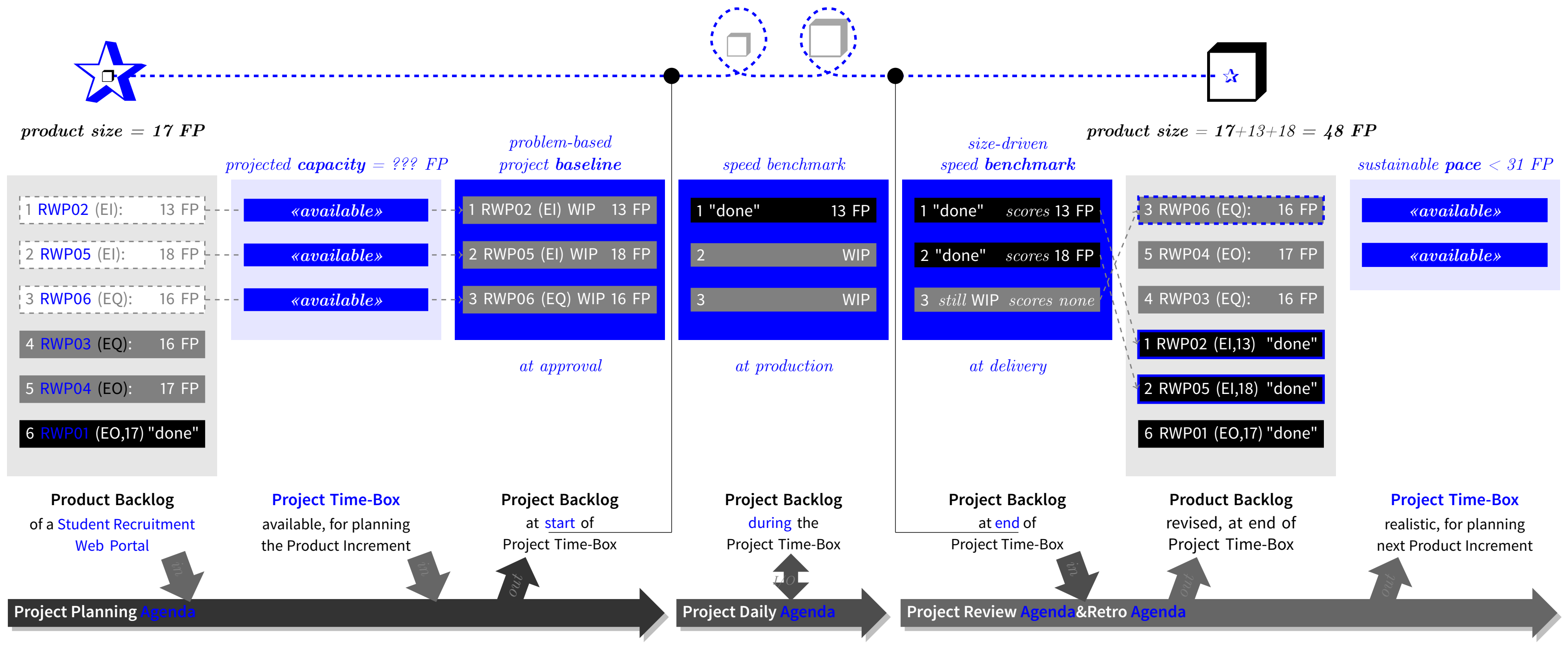
First, *the team inspects its Ongoing Performance* by comparing its function points planned in the beginning of the Project Time-Box with those actually scored at the end.

These facts are often made transparent and documented by a Sprint Report. ▶ Problem-Based Project Benchmarking eases the generation of such a report as it builds on a baseline made up of Requirements Work Packages, which are comparable among project (Time-Boxes and) Teams.

For instance, (variants of) requirements worked on in different project (Time-Boxe)s are comparable at the level of problems with each other. As these are equally scoped and in size, when using Requirements Work Packages, so is their assigned point value comparable, too. That way, the items of a problem-based project baseline and their speed benchmarks become meaningful across project (team)s. This empowers the Project Team to benefit from proven practices of others and to disseminate their lessons learned. The failure and success of realizing a specific solution for a specific problem can now be replayed and revised by any Project Team.

This *analysis of deviations from plan* by identifying its causes and developing countermeasures, is executed in the second activity of the Project Retro Agenda. In focus is the Patterns View of the "One4All" View Model of software architecture around which ► Problem-Based Project Adaptation takes place. The Project Team can try alternative patterns for gathering problems not "done" in this Project Time-Box or for designing a different solution to these. Project Backlog items whose problem-solution approach has proven its usefulness become recognizable units of best practices knowledge to the Project Team's lessons learned.

After completing the Project Retro Agenda, the Project Backlog is closed, and a new Project Time-Box can be started by executing the Project Planning Agenda again.



Legend:

EI, EQ, EO = External Input, External Inquiry, External Output RWP = Requirements Work Package

FP = Function Point WIP = work in progress

FIGURE 9.2 Executing an agile project process following A S.M.A.R.T. Scrum-A·Gen^{E}DA

9.4. Benchmarking a Problem-Based Project Baseline – A sustainable planning game

This section illustrates how to execute problem-based project planning by its sample application to a Student Recruitment Web Portal. Therefore, it follows four agendas of an agile project process as are elaborated by A S.M.A.R.T. Scrum-A·Gen^{E}DA.

In particular, it describes how the establishment of a problem-based project baseline and the measurement of related, size-driven speed benchmarks contribute to the comparability of projects' achievements, which is a prerequisite for implementing a sustainable project planning.

As shown in figure 9.2, the agendas for Project Planning, Project Daily, Project Review, and Project Retro engage with each other to manage project progress, which is documented by the Project Backlog, and inspectable at start, during, and at the end of a Project Time-box.

Planning the project according to the Project Planning Agenda takes a Product Backlog and a Project Time-Box as input for establishing the project baseline.

The output of this agenda is an approved project plan, which is given by a Project Backlog of sized and prioritized Requirements Work Packages (RWP), for which a do-able work plan of how to achieve desired project results is defined too.

The Product Backlog for the sample application of a Student Recruitment Web Portal in figure 9.2 consists of six Requirements Work Packages. Each includes one problem to be solved next. In Project Planning, the team must find a consensus on how to address these properly.

Therefore, the following activities **P 1.** to **P 3.** have to be executed:

EXAMPLE 9.1 A S.M.A.R.T. Scrum-A·Gen^{E}DA – Planning activities at the **start** of a project

P 1. Craft a Project Goal

As the Team has started to develop a MVP[a], all software functionality that belongs to the backbone of the desired product increment should be provided at first by the project. These are assumed to be those (20%) functional user requirements, which satisfy most (80%) users of this product. Thus the Project Goal from subsequent Project Time-Boxes is continued: *Deliver a MVP!*

P 2. Estimate and decide on Product Backlog items for the Project Time-Box

The selection of Product Backlog items to the Project Time-Box available creates the Project Backlog. It forms a baseline of planned product scope and size against which the completion of respective work items is compared. This comparison results in (speed) benchmarks for project(progres)s.
The top three Requirements Work Packages (RWP) as given in the outer left Product Backlog for the Student Recruitment Web Portal in figure 9.2 are at highest priority to be done in the available Project Time-Box for the delivery of a MVP.
As the functional size for each Requirements Work Package can be determined by ▶ Problem-Based Estimating, each work item of the Product Backlog is (reasonably assumed[b] to be already) assigned with a function point value.
As can be seen at the upper left corner of the Product Backlog in 9.2, the size delivered by the latest Product Increment for the Student Recruitment is 17 FP for providing RWP01 software functionality. If past performance of the team, i.e. its speed is known, it can be used for determining the projected capacity of a Project Time-Box. For this example, the Project Time-Box is of unknown size, since the Team is newly formed, and the problems to be solved are of a different kind (EI, EQ) compared to what is already implemented in the latest Product Increment (EO). The Team decides to select all three top RWP from the Product Backlog to become work in progress (WIP) that is to be completed during the project. For exploring how much they are really capable to do in a Project Time-Box, the team commits to the deliver(y speed of) 47 FP.

P 3. Plan how to deliver Product Backlog items into a "done" Increment

The Plans View of the "One4All" View Model of software architecture assists this activity. It enables the design of a solution alternative that fits the problem covered in each Requirements Work Package. By use of this ▶ Problem-Based Adaptation Frame·work, patterns guide the transition of desired software functionality to the delivered product increment through an instant blueprint sufficient for planning as well as acceptance[c] of the work needed to be done.

[a]MVP = Minimum Viable Product
[b]This is necessary to satisfy the first validation condition of this agenda.
[c]This is necessary to satisfy the second validation condition of this agenda.

Daily doings on the project work During the Project Time-Box the team develops the Product Increment. Change to the Product Backlog is not in the focus at this time in a project.

Project progress as made transparent by the Project Backlog is now of interest. The Project Daily Agenda is applicable for keeping it up-to-date during the elapsing Project Time-Box.

The Project Daily Agenda takes the Product Increment and the Project Backlog as input for monitoring which work items of the project baseline have already been "done" or are still work in progress (WIP).

The output of this agenda is a synchronization of the altered Project Backlog and the evolved Product Increment, which measures project progress during execution of the actual software production. This allows for timely intervention in case of deviations.

The **EXAMPLE 9.2** on page 190 presents the use of the Project Daily Agenda by executing its activities **D 1.** and **D 2.** for the Student Recruitment Web Portal case study.

EXAMPLE 9.2 A S.M.A.R.T. Scrum-A·Gen^{E}DA – Daily activities **during** a project

D 1. Do work on Project Backlog items

The Team has to implement the work plan for each Project Backlog item. In case of impediments, it has to strive for alternative solutions as described in **D 2.**

D 2. Inspect and adapt for project's work progress

The Project Backlog at production time is shown in the middle section of figure 9.2. At this point in the Project Time-Box, the Team has already completed RWP02, which is of highest priority to be "done" within this project. This success of being in the position to deliver desired software functionality scores the first 13 function points to the speed benchmark for this project. Respectively, RWP02 is marked as "done" in the ongoing Project Time-Box. This is necessary to satisfy the second validation condition of this agenda.
The Project Backlog items RWP05 and RWP06 remains as WIP to be done for the Product Increment of this Project Time-Box. The Team should not only make sure that project progress is in time, but also that it is in ongoing alignment[a] with the Project Goal.

[a]This is necessary to satisfy the first validation condition of this agenda.

Re-Vision of the project achievements The Project Review Agenda and Project Retro Agenda are both executed at the end of the Project Time-Box, so the input required is largely comparable. They take over the Project Backlog and the Product Increment as they are, after the time for the ongoing project has elapsed. No changes are made to these inputs at this time.

Of most interest is the Project Backlog at the end of the Project Time-Box as it drives the planning of subsequent projects for this product. It not only exhibits the progress made during production, it also embodies related lessons learned of “what works” [86] for the Team in the project while “mov[ing] towards a common understanding” [86] of (what is) the product (about). Accordingly, the Project Backlog not only contributes to the revision of the Product Backlog, which belongs to the output of these agendas too, it also helps to revise the capacity of the Project Time-Box, by which the Team is enabled to make more realistic commitments and consequently to proceed at a sustainable pace. The following activities **R 1.** to **R 4.** have to be executed:

EXAMPLE 9.3 A S.M.A.R.T. Scrum-A·Gen^{E}DA – Review&Retro activities at the **end** of a project

R 1. Inspect Product Increment

The team demonstrates the Product Increment to its key stakeholders, which exhibits the items of the Project Backlog that have been "done" in the Project Time-Box. For the example increment, the Team has to deal without software functionality for RWP06.

R 2. Adapt Product Backlog

The team adapts the Product Backlog based on the feedback provided by the key stakeholders, and supported by the Processes View as applicable by use of ▶ Problem-Based Project Adaptation. As a result of this discussion, the priority and presence of Requirements Work Packages in the Product Backlog is changed[a] and integrates the results as produced for each work item of the Project Backlog. For the example Product Backlog on the right side of figure 9.2 this means, the progress made for RWP02 and RWP05 results a reprioritization of these by moving them to the bottom of the Product Backlog, as they represent "done" software functionality, which is delivered in the Product Increment. RWP06 remains work in progress (WIP) as it is not implemented, and thus not part of the Product Increment. It moves to the top of the Product Backlog, since it is at highest priority compared with the other items of the Product Backlog after completion of RWP02 and RWP05. RWP03 and RWP04 are reprioritized[b] as (RWP04:) having the candidate’s data available in a PDF file is of more value to the admin users of the students’s recruitment process, than (RWP03:) the simple presentation of a candidate’s data on the screen only.

[a]This is necessary to satisfy the first and second validation condition of the Project Review Agenda.
[b]Note that this is an example only. Of course, other reasons can be found for this decision making.

EXAMPLE 9.4 A S.M.A.R.T. Scrum-A·Gen^{E}DA – Review&Retro activities , continued

R 3. Inspect Team Performance

As illustrated in figure 9.2 the speed benchmark of 31 FP is scored by the Team for the project baseline. Remaining WIP from the project score no function points to this speed benchmark the Team has earned in the Project Time-Box . Since this size-driven performance measurement implements a ▶ Problem-Based Project Benchmarking, it serves the reasonable limitation of the Project Time-Box in subsequent project planning[a], and supports the Team in identifying alternative solutions for those desired software functionality, which has not been delivered by the Product Increment. Past performance becomes a reusable knowledge base for seeking ways out of underarchievement[b] as is covered by activity **R 4.**

R 4. Adapt for Team Improvement

As all items of the Project Backlog renew the Product Backlog[c], so are the team's lessons learned at the end of the project.
By use of ▶ Problem-Based Project Adaptation their problem solving knowledge is made reusable via the Patterns View of the One4All View Model.
Work items successfully done represent good practices, whereas those not completed can be remembered as failed attemps.
Since these valuable experiences gathered by the team are enshrined in recognizable units of software functionality, i.e. Requirements Work Packages, they become a reusable means for other projects or teams to compare their project achievements and improve their practices.
For instance, as the Product Backlog in figure 9.2 does not provide any "done" Requirements Work Packages for EQ-problems, the team could check other products for implementations of this kind of problems, such that they get an idea of what works (realistically) to get RWP06 and RWP03 done.
As patterns are the medium used in problem-based project planning for knowledge gathering and sharing, these create useful synergies for projects and their teams. It is this pattern-enhanced capability of project teams to adopt the problem-solving knowledge of others, which lets them adapt their project practices and thereby speed up project progress.

[a]This fulfills the second validation condition of the Project Retro Agenda
[b]This fulfills the first validation condition of the Project Retro Agenda
[c]This satisfies the first validation condition of Project Review Agenda

9.5. Discussion & Related Work

A problem-based project baseline as introduced in this chapter not only supports size-driven speed benchmarks, which make *performance measurement* of the software delivery process for productivity analysis [178, page 368] possible.

It also allows for the implementation of a pattern-enhanced *improvement process* by best practice benchmarking [113, page 1 and page 7], which is designed for determining the proven problem-solving knowledge, that has enabled and can be reused for establishing comparable project achievements.

To this extent, the proposed approach of Problem-Based Project Benchmarking is in line to "benchmarking as a key component of quality assurance and process improvement (Watson, 1992)." [71, page 21], as well as to "Continuous engineering [which] is about strategic reuse, about not re-inventing the wheel and about actively preventing re-work." [161]

As "Benchmarking processes are not easy to implement [...as they] will be successful only if made an integral part of the project process" [71, page 24 and 25], problem-based project benchmarking as developed in this chapter is integrated to a Scrum-alike project process named A S.M.A.R.T. Scrum-A·Gen^{E}DA. It copes successfully with **known barriers** to the introduction of benchmarking processes, such as

1. ensuring data validity, addressing
2. the fear of data sharing, and
3. the lack of slack for data collection.

Therefore, **problem-based benchmarking takes these known barriers into account** by utilizing

1. a pattern-based measurement and development process, which cares by design for data completeness and accuracy, as it is in accordance with IFPUG ISO standard, instead of relying exclusively on gutfeel assessment. "Benchmarking is the most credible of all justifications for operations [...] as it "removes the subjectivity from decision making" [48, page 15].
2. "an identity-blind [since problem-based project benchmarking] process, whereas data are posted without attribution" [71, page 24] for mitigating the concerns of unwanted exposure to competitive advantages or personal weaknesses, especially given that "measuring employees [...] on an individual basis is illegal in some countries." [15, page 465] For instance, performance and behavioral control may be in conflict with the work constitution law, the general data protection regulations, or personal rights, which belong to the fundamental rights in the German Constitution.
3. the items of product and project backlog, which are already in place, i.e. Requirements Work Packages as structured reference points for the data harvesting and storage. That way, no additional consumption of time or resources for the benchmarking process occurs, as respective intelligence gathering based on benchmarking data comes along with executing the project (planning) process itself. It applies these multi-purpose units for the delivery and comparison of satisfied user expectations as well.

9.6. Summary

This chapter presents a Scrum-alike project process, which enables problem-based project planning and benchmarking.

This project process is documented by A S.M.A.R.T. Scrum-A·Gen^{E}DA, which implements agile practices for the continuous improvement of product management and product engineering perspectives in software development projects. It is structured into four agendas, of which each adopts exactly one event of a Scrum Sprint. In addition, it integrates Problem-Based Project Estimating and Problem-Based Project Adaptation to these agendas too, which makes problem-based project benchmarking of the work "done" in a project timebox possible.

One these grounds, requirements work packages as developed in section 5.3 of this dissertation becomes ultimately S.M.A.R.T. Their use as units for the definition of **s**pecific and **m**easurable product scope, as well as their use as units for the design of **a**chievable production work plans, is further developed for their use as units for the **r**ealistic and **t**ime-bound benchmarking of project progress. That way, a project baseline (sized product scope) together with its related performance measurement (speed of production), which both build on these units, become comparable among projects and teams.

Problem-Based Project Benchmarking takes the speed of delivering a working product increment into systematic account. It factors in the team's need to know its speed, which is used as input to their project planning, i.e. for making credible commits. Speed is given by functions points in Problem-based Project Benchmarking, which are achieved by the team for completed items of a project work plan in a project timebox. This work plan is set up by a project backlog of requirements work packages, which all are of known functional size. Consequently, it takes account of proven progress (scored point values for solved problems) instead of anticipated progress (estimated point values based on a poker game) against a recognizable project plan. That way, problem-based project benchmarking establishes a sustainable planning game due to the provision and use of (functional) size (as performance) data.

As the satisfaction of user expectations is bound to problems and not to people, implemented solutions and involved success benchmarks become available as sharable lessons learned for other projects. These patterns of experienced problem-solving knowledge, and their related point values assist teams in establishing feasible work plans that fit to a defined project timebox.

Part V.

Case Studies

Part V Case Studies presents two comprehensive application examples for illustrating the use and importance of the contributions given by this dissertation, to control speed of software projects. Chapter 10 Vacation Rentals Web Application revisits a didactic play from the lecture Software Technology, which is used by the Working Group Software Engineering at the University of Duisburg-Essen to demonstrate the ADIT procedure, an agenda-driven and pattern-based software development process. Chapter 11 Student Recruitment Web Portal applies the contributions of this dissertation to a software application, which has emerged from a student project and is used by the Faculty of Engineering at the University of Duisburg-Essen in support of their "International Studies in Engineering" program.

10. Vacation Rentals Web Application

The vacation rentals is an online travel agency, where some customers or respective guests can go browsing holiday offers, make bookings, etc. The travel agent or respective staff member is responsible for maintaining the holiday offers, that are available via the vacation rentals web application.

This section revisits a respective software requirements documentation as given in [67, 108], for illustrating the application of problem-based project estimating as introduced in chapter 6 Problem-Based Estimating Method, as well as problem-based project adaptation as defined in chapter 8 Problem-Based Adaptation Framework to this case study.

Thereby, this chapter provides the basis for discussing the use, changes, and impact made by the contributions of this dissertation on this specific software application, and in general to software engineering activities.

10.1. Requirements Decomposition

The vacation rentals web application is based on nine requirements, which are given in table 10.1. These are decomposed to seven independent problems by means of problem frames.

No.	Requirement Statement	Problem Name
R01	A staff member can **make** holiday offers *available*.	Make
R02	A guest can **browse** *available* holiday offers.	Browse[Offerings]
R03	A guest can **book** *available* holiday offers, which then are *reserved*.	Book
R04	After a guest **book**s a holiday offer, she is provided a *corresponding* invoice.	
R05	If a *reserved* holiday offer is not paid within 14 days, it is automatically set *available* again.	Reset
R06	A staff member can **record** when a payment is received[…, which makes a holiday offer *booked*].	Pay
R07	A staff member can **rate** the status of a vacation home, after a guest left it.	Rate
R08	If the status of a vacation home is rated negatively, the guest receives an *additional* invoice.	
R09	A staff member can **browse** *reserved*[…and *booked*] holiday offers.	BrowseBookings

TABLE 10.1 Requirements for a vacation rentals web application

For the sake of clarity, the requirement statements for R06 and R09 have been slightly modified, as well as the problem name for R02. The constituent parts of each requirement statement, which relate to a problem domain are framed by a colored text box, the ones that relate to a **causal** phenomenon are written in boldface and the ones that relate to *symbolic* phenomena are in italics.

Due to the application of problem-based functional size measurement patterns and the criteria for joining measurable problems, these nine requirements are (re-)decomposed to only three, unique requirements work packages: RWP01, RWP02, and RWP03, which are summarized in table 10.2.

No.	Problem Name	Requirement Work Package
R09	BrowseBookings	RWP02: Present Holiday Offers
R02	Browse[Offerings]	
R07	Rate	RWP03: Provide Invoice
R08	Rate	
R04	Book	
R03	Book	RWP01: Prepare Holiday Offer
R05	Reset	
R06	Pay	
R01	Make	

TABLE 10.2 Decomposition of requirements for a vacation rentals web application

All the details on the setup of each requirements work package are elaborated in the following.

10.1.1. Problem description for RWP01: Prepare Holiday Offer

The requirements R01, R03, R05, and R06 of the vacation rentals web application that constitute the Requirements Work Package 01: Prepare Holiday Offer (RWP01) given in table 10.3 are originally decomposed to different independent problems, namely $Make$, $Book$, $Reset$, and Pay by instantiating the $update$ or the $simple\ transformation$ problem frame as documented in [108, slide 257], which are no Basic FSM patterns. Now, these four requirements belong to the same measurable problem.

No.	Problem Name	Problem Frame	Problem Class
R03	Book	update variant	simple workpieces (external input, EI)
R05	Reset	simple transformation	
R06	Pay	update	
R01	Make	update	

TABLE 10.3 Setup of Requirements Work Package RWP01: Prepare Holiday Offer

Reconsidering the requirements' problem class by problem-based functional size measurement patterns as given in table 5.5 results in classifying their type of functionality to one which is concerned with processing received information, i.e. to TOFF-i. as introduced in section 5.4.3.

In other words, the primary intent shared by the requirements R01, R03, R05, and R06 is to take received information from the guest or staff member for preparing to some extent a holiday offer. This is the concern, which separates relevant from irrelevant requirements for any requirements work package, and the reason why R01, R03, R05, and R06 become members in a common requirements work package representing an instance of the $simple\ workpieces$ problem frame. Thus, RWP01 implements a measurable problem, which belongs to an external input (EI).

In this connection, it is not ignored that preparing holiday offers must be presented to the users in some way, which may initially have given the reason to make use of the $update$ problem frame. Nevertheless, this is not at the heart of the problems that are implemented in this requirements work package and to be solved here. RWP01 is about preparing holiday offers, which is also a recognizable characteristic in its specification given in figure 10.5.

Of special interest is the assignment of R05 to this requirements work package, which is due to two reasons that are in contrast to the original documentation of the vacation rentals. First, R05 is a level I. micro problem according to table 5.3 on page 55, which is now mounted to the basic level II. of requirements decomposition, that provides more problem context. Second, R05 newly shares the phenomenon $book$ with R03 as illustrated in figure 10.5.

What happened is that the $Book$ problem R03 integrates the former $Reset$ problem R05 to a common problem scope, which is justified by the criteria given on page 66 that help to identify overlaps in the problem scope of measurable problems, and thereby explicating their logical relation or the involved requirements dependencies.

R01, R03, R05, and R06 fulfill all three criteria: 1st. they fit the same problem class with respect to $holiday\ offer$, 2nd. they have overlaps in their problem scope with respect to status information $available$, $reserved$, and $booked$, as well as $book$, which do not let them change independently of each other, and 3rd. they are in a parallel composition with the $Reset\ Problem$ according to the life-cycle for the vacation rentals, which makes them candidates for merger to one common measurable problem. These four requirements form a coherent set of software functionality that is reasonably implemented in one requirements work package. Their overlap is a common constraint on the $holiday\ offer$ problem domain.

10.1.2. Problem description for RWP02: Present Holiday Offers

The requirements R02 and R09 of the vacation rentals web application are implemented in the Requirements Work Package 02: Present Holiday Offers (RWP02) given in table 10.4. In the original requirements decomposition as documented by [108, slide 257], they instantiate different independent problems, namely $BrowseOfferings$ and $BrowseBookings$ by use of the $query$ problem frame, which is a Basic FSM patterns as defined in table 5.5.

No.	Problem Name	Problem Frame	Problem Class
R02	BrowseBookings	query	query (external inquiry, EQ)
R09	BrowseOfferings	query	

TABLE 10.4 Setup of Requirements Work Package RWP02: Present Holiday Offers

The primary intent shared by the requirements R02 and R09 is to retrieve information from the given holiday offers and to show these to a user via the web browser, i.e. their type of functionality belongs to TOFF-ii. as introduced in section 5.4.3. This concern is the reason why R02 and R09 become members in a common requirements work package RWP02, keeping them as instances of the $query$ problem frame. They describe variants of finally the same measurable problem, one that coincides with an external inquiry as illustrated in figure 10.6. The criteria given on page 66 provide additional justification for implementing R02 together with R09 in RWP02.

10.1.3. Problem description for RWP03: Provide Invoice

The requirements R04, R07 and R08 of the vacation rentals web application are jointly implemented in the Requirements Work Package 03: Provide Invoice (RWP03) given in table 10.5. In the original requirements decomposition as documented by [108, slide 257], they instantiate different independent problems, namely $Rate$ for R08 and R07, and the $BrowseBookings$ problem for R04 making use of the $update$ variant problem frames, which are no Basic FSM patterns.

No.	Problem Name	Problem Frame	Problem Class
R04	Book	update	commanded data-based control (external output, EO)
R07	Rate	update variant	
R08	Rate	update variant	

TABLE 10.5 Setup of Requirements Work Package RWP03: Provide Invoice

Comparable to RWP01, which is described on page 199, RWP03 comprises multi-composite $update$ problems, which must be reduced to a single concern in order to fit a basic FSM pattern, and thus to represent a measurable problem.

In contrast to RWP01, whose type of involved software functionality was classified as TOFF-i. that coincides with an external input, this time the primary intent considered by RWP03 is reduced to the concern of processing derived information, that is TOFF-iii, which coincides with an external output as introduced in section 5.4.3. This type of functionality is more appropriate for characterizing the measurable problem at hand.

As figure 10.7 illustrates, with regard to the user triggering a $rate$ or $book$, RWP03 creates invoices based on some derived information for a $holiday\ offer$ and sends these to a respective guest via email. This is best covered by the problem-based functional size measurement pattern #15 commanded data-based control as given in table 5.5. The criteria given on page 66 provide additional justification for implementing R04, R07 and R08 in RWP03.

10.2. Requirements Measurement

This chapter gives in its sections 10.2.1 to 10.2.3 a problem-based functional size measurement of the Vacation Rentals Web Application by following the Frame Counting Agenda.

10.2.1. Problem count for RWP01: Prepare Holiday Offer

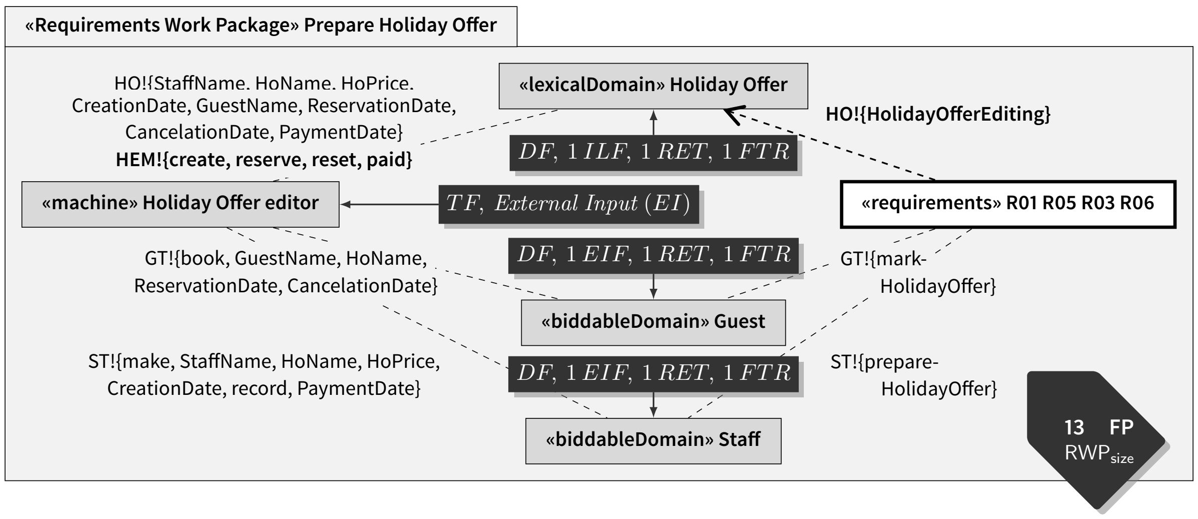

«Requirements Work Package» Prepare Holiday Offer
HO!{StaffName, HoName, HoPrice, CreationDate, GuestName, ReservationDate, CancelationDate, PaymentDate}
HEM!{create, reserve, reset, paid}
«lexicalDomain» Holiday Offer
HO!{HolidayOfferEditing}
DF, 1 ILF, 1 RET, 1 FTR
«machine» Holiday Offer editor
TF, External Input (EI)
«requirements» R01 R05 R03 R06
DF, 1 EIF, 1 RET, 1 FTR
GT!{book, GuestName, HoName, ReservationDate, CancelationDate}
GT!{mark-HolidayOffer}
«biddableDomain» Guest
ST!{make, StaffName, HoName, HoPrice, CreationDate, record, PaymentDate}
DF, 1 EIF, 1 RET, 1 FTR
ST!{prepare-HolidayOffer}
«biddableDomain» Staff
13 FP
RWP size

FIGURE 10.1 Determining the functional size of RWP01: Prepare Holiday Offer

Comments on counting process activity	Results of activity
1. Classify FUR by Functional Size Measurement Patterns.	
Applied validation conditions: V.i - V.iii	PF 2.7 simple workpieces problem
2. Determine Data Functions. 2.a Identify problem domains as data functions.	
Applied validation conditions: V.iv, (V.v)	3 data functions: Staff, Guest, Holiday Offer
2.b Classify data functions into ILF or EIF.	
Applied validation conditions: V.vi, V.vii, (V.viii,) V.ix, V.x	2 EIF: Staff, Guest 1 ILF: Holiday Offer
2.c Count DET for each data function.	
Applied validation conditions: V.xi	$DET_{Staff} = 5$ $DET_{Guest} = 4$ $DET_{Holiday\ Offer} = 8$
2.d Count RET for each data function.	
Applied validation conditions: V.xii	$RET_{Staff} = 1$ $RET_{Guest} = 1$ $RET_{Holiday\ Offer} = 1$
2.e Determine functional complexity for data functions.	
Applied validation conditions: V.xiii	$ILF_{DET} = 8$
Applied validation conditions: V.xiv	~~$EIF_{DET} = 5 + 4 = 9$~~
Applied validation conditions: V.xv(, V.xvi)	$\mathbf{EIF_{DET}} = 9 - 5 - 4 = \mathbf{0}$
Applied validation conditions: V.xvii	$ILF_{RET} = 1$
Applied validation conditions: V.xviii	$EIF_{RET} = 2$
Applied validation conditions: V.xix(,V.xxi) due $EIF_{DET} = 0$	$EIF_{Complexity} = \{n/a\}$

Comments on counting process activity	Results of activity
Applied validation conditions: **V.xx** by $ILF_{Complexity}(1,8) = low$	$ILF_{Complexity} = \{low\}$
2.f Determine functional size for data functions.	
$ILF_{Size}(ILF_{Complexity}, ILF) = ILF_{Size}(low, ILF) = 7$ $EIF_{Size}(EIF_{Complexity}, EIF) = EIF_{Size}(\{n/a\}, EIF) = 0$ Applied validation conditions: **V.xxii**, **V.xxiii**(, **V.xxiv**)	ILF_{Size} = 7 function points EIF_{Size} = 0 function points
3. Determine Transactional Function. 3.a Identify machine domain as transactional function.	
Applied validation conditions: **V.xxv**	TF: Holiday Offer editor
3.b Classify transactional function as either EI, EQ, or EO.	
Applied validation conditions: for *simple workpieces* **V.xxvi**	$TF_{Type} = EI$
3.c Count FTR for transactional function.	
TF_{FTR} = n ILF + m EIF = 1 + 2 = 3. Applied validation conditions: **V.xxvii**	$TF_{FTR} = 3$
3.d Count DET for transactional function.	
Applied validation conditions: **V.xxviii**, **V.xxix**, **V.xxx**, **V.xxxi**	$TF_{DET} = 12$
3.e Determine functional complexity for transactional function.	
$TF_{Complexity}(TF_{Type}, TF_{FTR}, TF_{DET}) = TF_{Complexity}(EI, 3, 12)$ Applied validation conditions: **V.xxxii**	$TF_{Complexity} = high$
3.f Determine functional size for transactional function.	
$TF_{Size}(TF_{Complexity}, TF_{Type}) = TF_{Size}(high, EI)$ Applied validation conditions: **V.xxxiii**	TF_{Size} = 6 function points
4. Report Functional Size for FUR.	
$MeasurableProblem_{size} = ILF_{Size} + EIF_{Size} + TF_{Size} = 7 + 0 + 6$ Applied validation conditions: **V.xxxiv**	$RWP01_{size} =$ **13 function points**

10.2.2. Problem count for RWP02: Present Holiday Offers

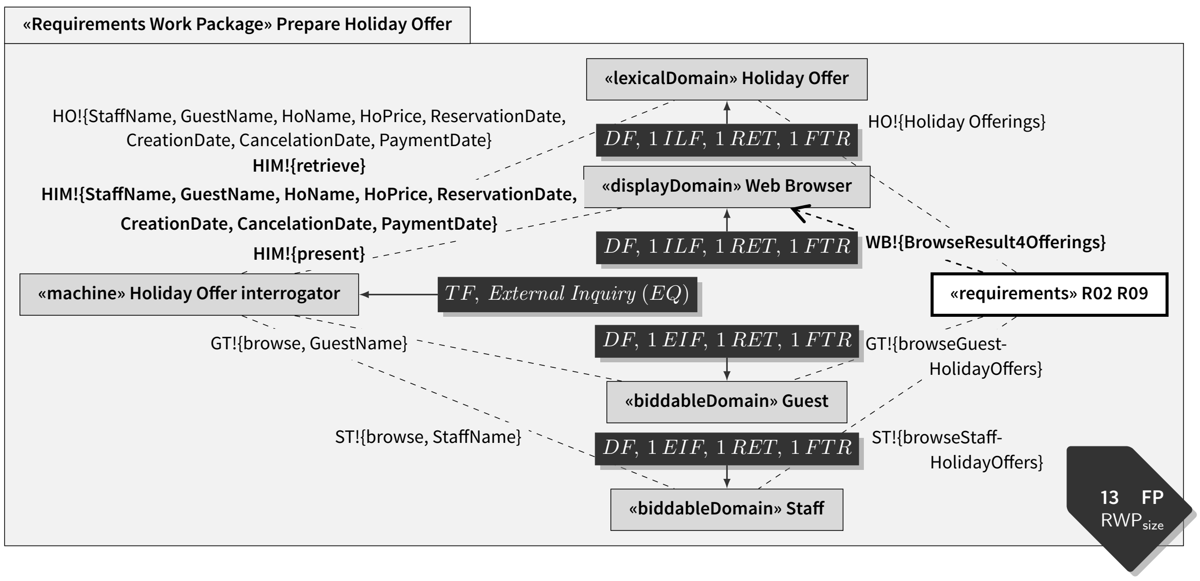

«Requirements Work Package» Prepare Holiday Offer
«lexicalDomain» Holiday Offer
HO!{StaffName, GuestName, HoName, HoPrice, ReservationDate, CreationDate, CancelationDate, PaymentDate}
HO!{Holiday Offerings}
DF, 1 ILF, 1 RET, 1 FTR
HIM!{retrieve}
HIM!{StaffName, GuestName, HoName, HoPrice, ReservationDate, CreationDate, CancelationDate, PaymentDate}
«displayDomain» Web Browser
HIM!{present}
DF, 1 ILF, 1 RET, 1 FTR
WB!{BrowseResult4Offerings}
«machine» Holiday Offer interrogator
TF, External Inquiry (EQ)
«requirements» R02 R09
DF, 1 EIF, 1 RET, 1 FTR
GT!{browse, GuestName}
GT!{browseGuest-HolidayOffers}
«biddableDomain» Guest
ST!{browse, StaffName}
DF, 1 EIF, 1 RET, 1 FTR
ST!{browseStaff-HolidayOffers}
«biddableDomain» Staff
13 FP
RWPsize

FIGURE 10.2 Determining the functional size of RWP02: Present Holiday Offers

Comments on counting process activity	Results of activity
1. Classify FUR by Functional Size Measurement Patterns.	
Applied validation conditions: V.i - V.iii	PF 3.8 query problem
2. Determine Data Functions. 2.a Identify problem domains as data functions.	
Applied validation conditions: V.iv, (V.v)	4 data functions: Staff, Guest, Holiday Offer, Web Browser
2.b Classify data functions into ILF or EIF.	
Applied validation conditions: V.vi, V.vii, (V.viii,) V.ix, V.x	2 EIF: Staff, Guest 2 ILF: Holiday Offer, Web Browser
2.c Count DET for each data function.	
Applied validation conditions: V.xi	$DET_{Staff} = 1$ $DET_{Guest} = 1$ $DET_{Holiday\ Offer} = 8$ $DET_{Web\ Browser} = 8$
2.d Count RET for each data function.	
Applied validation conditions: V.xii	$RET_{Staff} = 1$ $RET_{Guest} = 1$ $RET_{Holiday\ Offer} = 1$ $RET_{Web\ Browser} = 1$
2.e Determine functional complexity for data functions.	
Applied validation conditions: V.xiii	~~$ILF_{DET} = 8 + 8 = 16$~~
Applied validation conditions: V.xvi	$\mathbf{ILF_{DET}} = 16 - 8 = \mathbf{8}$
Applied validation conditions: V.xiv	~~$EIF_{DET} = 1 + 1 = 2$~~
Applied validation conditions: V.xv	$\mathbf{EIF_{DET}} = 2 - 1 - 1 = \mathbf{0}$
Applied validation conditions: V.xvii	$ILF_{RET} = 2$
Applied validation conditions: V.xviii	$EIF_{RET} = 2$
Applied validation conditions: V.xix(,V.xxi) due $EIF_{DET} = 0$	$EIF_{Complexity} = \{n/a\}$
Applied validation conditions: V.xx by $ILF_{Complexity}(2, 8) = low$	$ILF_{Complexity} = \{low\}$
2.f Determine functional size for data functions.	
$ILF_{Size}(ILF_{Complexity}, ILF) = ILF_{Size}(low, ILF) = 7$ $EIF_{Size}(EIF_{Complexity}, EIF) = EIF_{Size}(\{n/a\}, EIF) = 0$ Applied validation conditions: V.xxii, V.xxiii(, V.xxiv)	ILF_{Size} = 7 function points EIF_{Size} = 0 function points

Comments on counting process activity	Results of activity
3. Determine Transactional Function.	
3.a Identify machine domain as transactional function.	TF: Holiday Offer interrogator
3.b Classify transactional function as either EI, EQ, or EO.	
Applied validation conditions: for $query$ **V.xxv**	$TF_{Type} = EQ$
3.c Count FTR for transactional function.	
TF_{FTR} = n ILF + m EIF = 2 + 2 = 4. Applied validation conditions: **V.xxvi**	$TF_{FTR} = 4$
3.d Count DET for transactional function.	
Applied validation conditions: **V.xxvii**, **V.xxviii**, **V.xxix**, **V.xxx**, **V.xxxi**	$TF_{DET} = 13$
3.e Determine functional complexity for transactional function.	
$TF_{Complexity}(TF_{Type}, TF_{FTR}, TF_{DET}) = TF_{Complexity}(EQ, 4, 13)$ Applied validation conditions: **V.xxxii**	$TF_{Complexity} = high$
3.f Determine functional size for transactional function.	
$TF_{Size}(TF_{Complexity}, TF_{Type}) = TF_{Size}(high, EQ)$ Applied validation conditions: **V.xxxiii**	$TF_{Size} =$ 6 function points
4. Report Functional Size for FUR.	
$MeasurableProblem_{size} = ILF_{Size} + EIF_{Size} + TF_{Size} = 7 + 0 + 6$ Applied validation conditions: **V.xxxiv**	$RWP02_{size} =$ **13 function points**

10.2.3. Problem count for RWP03: Provide Invoice

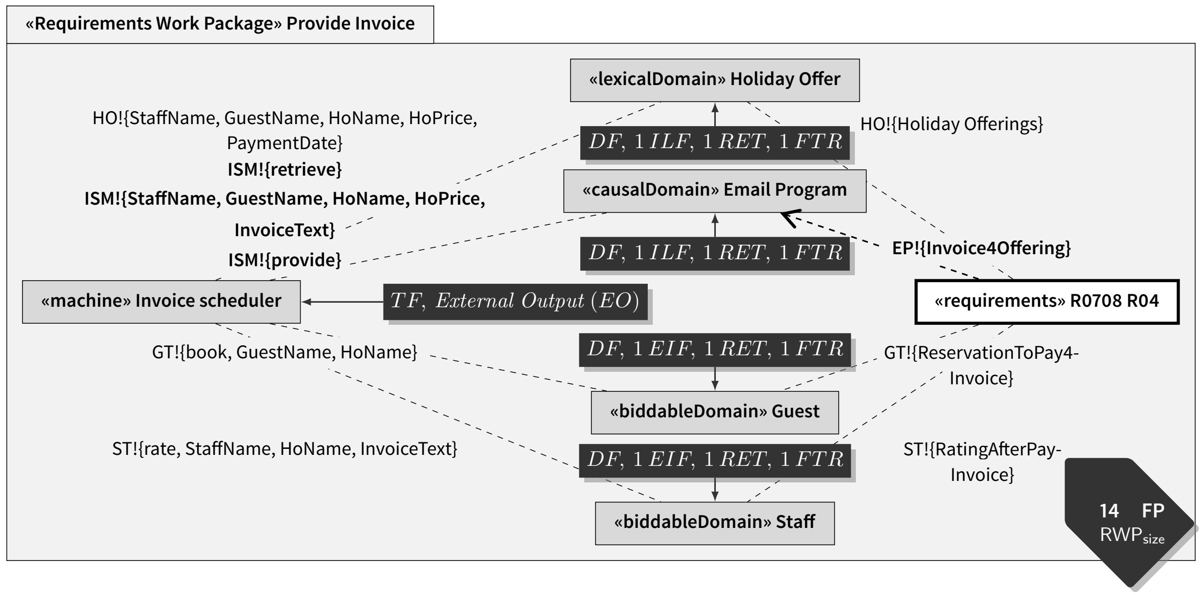

«Requirements Work Package» Provide Invoice
«lexicalDomain» Holiday Offer
HO!{StaffName, GuestName, HoName, HoPrice, PaymentDate}
ISM!{retrieve}
ISM!{StaffName, GuestName, HoName, HoPrice, InvoiceText}
ISM!{provide}
HO!{Holiday Offerings}
DF, $1\ ILF$, $1\ RET$, $1\ FTR$
«causalDomain» Email Program
DF, $1\ ILF$, $1\ RET$, $1\ FTR$
EP!{Invoice4Offering}
«machine» Invoice scheduler
TF, $External\ Output\ (EO)$
«requirements» R0708 R04
GT!{book, GuestName, HoName}
DF, $1\ EIF$, $1\ RET$, $1\ FTR$
GT!{ReservationToPay4-Invoice}
«biddableDomain» Guest
ST!{rate, StaffName, HoName, InvoiceText}
DF, $1\ EIF$, $1\ RET$, $1\ FTR$
ST!{RatingAfterPay-Invoice}
«biddableDomain» Staff
14 FP
RWP$_{size}$

FIGURE 10.3 Determining the functional size of RWP03: Provide Invoice

Comments on counting process activity	Results of activity
1. Classify FUR by Functional Size Measurement Patterns.	
Applied validation conditions: V.i - V.iii	PF 3.7 commanded data-based control problem
2. Determine Data Functions.	
2.a Identify problem domains as data functions.	
Applied validation conditions: V.iv, (V.v)	4 data functions: Staff, Guest, Holiday Offer, Email Program
2.b Classify data functions into ILF or EIF.	
Applied validation conditions: V.vi, V.vii, (V.viii,) V.ix, V.x	2 EIF: Staff, Guest 2 ILF: Holiday Offer, Email Program
2.c Count DET for each data function.	
Applied validation conditions: V.xi	$DET_{Staff} = 3$ $DET_{Guest} = 2$ $DET_{Holiday\ Offer} = 5$ $DET_{Email\ Program} = 5$
2.d Count RET for each data function.	
Applied validation conditions: V.xii	$RET_{Staff} = 1$ $RET_{Guest} = 1$ $RET_{Holiday\ Offer} = 1$ $RET_{Email\ Program} = 1$
2.e Determine functional complexity for data functions.	
Applied validation conditions: V.xiii	~~$ILF_{DET} = 5 + 5 = 10$~~
Applied validation conditions: V.xvi	$\mathbf{ILF_{DET}} = 10 - 4 = \mathbf{6}$
Applied validation conditions: V.xiv	~~$EIF_{DET} = 3 + 2 = 5$~~
Applied validation conditions: V.xv	$\mathbf{EIF_{DET}} = 5 - 2 - 3 = \mathbf{0}$
Applied validation conditions: V.xvii	$ILF_{RET} = 2$
Applied validation conditions: V.xviii	$EIF_{RET} = 2$
Applied validation conditions: V.xix(,V.xxi) due $EIF_{DET} = 0$	$EIF_{Complexity} = \{n/a\}$
Applied validation conditions: V.xx by $ILF_{Complexity}(2, 6) = low$	$ILF_{Complexity} = \{low\}$
2.f Determine functional size for data functions.	
$ILF_{Size}(ILF_{Complexity}, ILF) = ILF_{Size}(low, ILF) = 7$ $EIF_{Size}(EIF_{Complexity}, EIF) = EIF_{Size}(\{n/a\}, EIF) = 0$ Applied validation conditions: V.xxii, V.xxiii(, V.xxiv)	ILF_{Size} = 7 function points EIF_{Size} = 0 function points

Comments on counting process activity	Results of activity
3. Determine Transactional Function.	
3.a Identify machine domain as transactional function.	TF: Invoice scheduler
3.b Classify transactional function as either EI, EQ, or EO.	
Applied validation conditions: for $query$ **V.xxv**	$TF_{Type} = EO$
3.c Count FTR for transactional function.	
TF_{FTR} = n ILF + m EIF = 2 + 2 = 4. Applied validation conditions: **V.xxvi**	$TF_{FTR} = 4$
3.d Count DET for transactional function.	
Applied validation conditions: **V.xxvii**, **V.xxviii**, **V.xxix**, **V.xxx**, **V.xxxi**	$TF_{DET} = 13$
3.e Determine functional complexity for transactional function.	
$TF_{Complexity}(TF_{Type}, TF_{FTR}, TF_{DET}) = TF_{Complexity}(EO, 4, 13)$ Applied validation conditions: **V.xxxii**	$TF_{Complexity} = high$
3.f Determine functional size for transactional function.	
$TF_{Size}(TF_{Complexity}, TF_{Type}) = TF_{Size}(high, EO)$ Applied validation conditions: **V.xxxiii**	$TF_{Size} =$ 7 function points
4. Report Functional Size for FUR.	
$MeasurableProblem_{size} = ILF_{Size} + EIF_{Size} + TF_{Size} = 7 + 0 + 7$ Applied validation conditions: **V.xxxiv**	$RWP03_{size} =$ **14 function points**

10.3. Use Case Decomposition

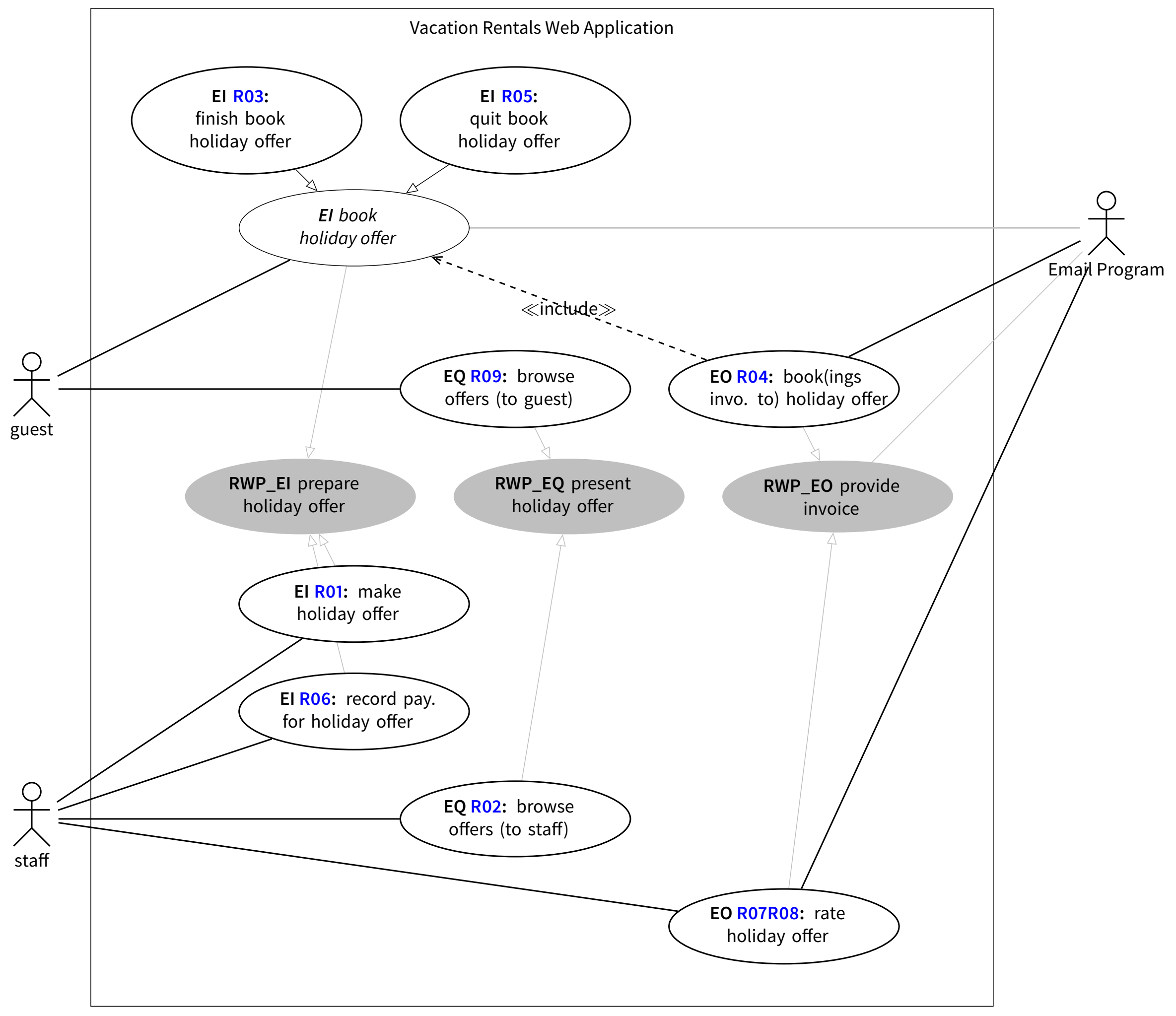
Figure 10.4 gives a use case diagram as an alternative requirements decomposition to the problem-based decomposition in table 10.2. As already illustrated and discussed for the use case decomposition of the Student Recruitment Web Portal in section 8.6 Sample Application to Use Case Decomposition on page 168, problem-based functional size measurement patterns are applicable for the identification and structuring of requirements into user stories as well as in UML use cases.

The benefits of this combination are multifold:

1. More flexibility in documenting the requirements.
2. Higher consistency between different requirement models.
3. Wider applicability and comparability of function point values, which result from problem-based functional size measurement by the Frame Counting Agenda.

   For instance: As figure 10.4 shows by the gray-colored generalization relations, use cases, which can be combined into one requirements work package (due to their shared referenced problem domain and problem class) become recognizable. This ensures that these are counted the same way and without redundancy.
4. Other requirement models can take advantage of the requirements synchronization approach as part of the Processes View in the "One4All" architectural view model, which is introduced in section 8.3. It allows for analyzing requirements dependency based on state transition diagrams, such as introduced in section 8.5 Synchronizing Requirements by a State Transition Diagram.

In figure 10.4, notational elements in gray serve only illustrative purposes. For instance, the inheritance relations to use cases **RWP_*** should show, that respective child use cases share the same underlying problem class. All other notational elements are conform to the OMG specification for UML [168].



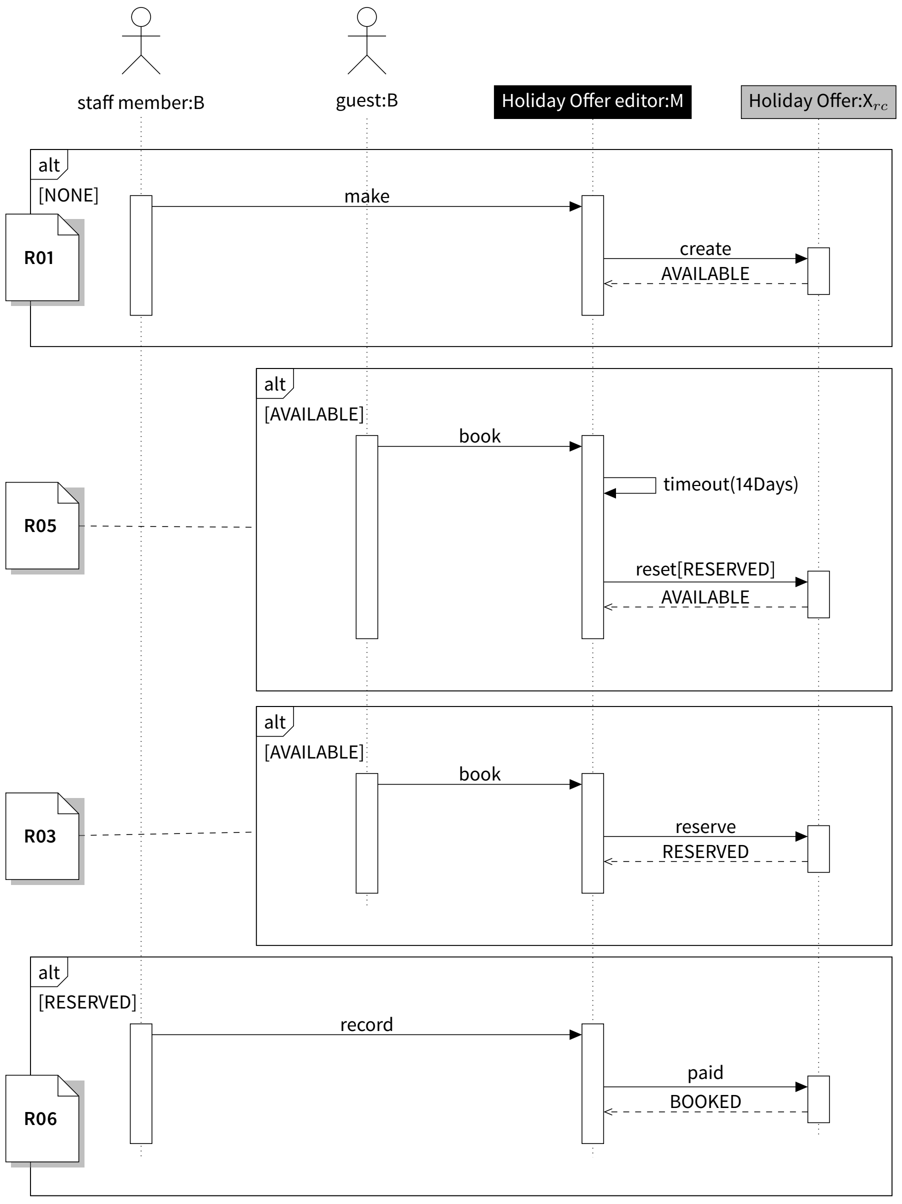
Vacation Rentals Web Application
EI R03: finish book holiday offer
EI R05: quit book holiday offer
EI book holiday offer
≪include≫
EQ R09: browse offers (to guest)
EO R04: book(ings invo. to) holiday offer
RWP_EI prepare holiday offer
RWP_EQ present holiday offer
RWP_EO provide invoice
EI R01: make holiday offer
EI R06: record pay. for holiday offer
EQ R02: browse offers (to staff)
EO R07R08: rate holiday offer
guest
staff
Email Program

FIGURE 10.4 UML use case diagram for the Vacation Rentals Web Application

10.4. Requirements Specification

This section lists all task scenarios (as introduced and applicable for Problem templates in chapter 7.4) for the requirements **R01** to **R08** that belong to the requirements work packages in the Vacation Rentals case study.

10.4.1. Task scenarios of RWP01: Prepare Holiday Offer



Legend: Objects are instances of a (b)iddable, le(x)ical, or (m)achine domain.
The subscript rc indicates the problem domain to which a requirements constraint (rc) refers to.

FIGURE 10.5 Specification of Requirements Work Package RWP01: Prepare Holiday Offer

10.4.2. Task scenarios of RWP02: Present Holiday Offers

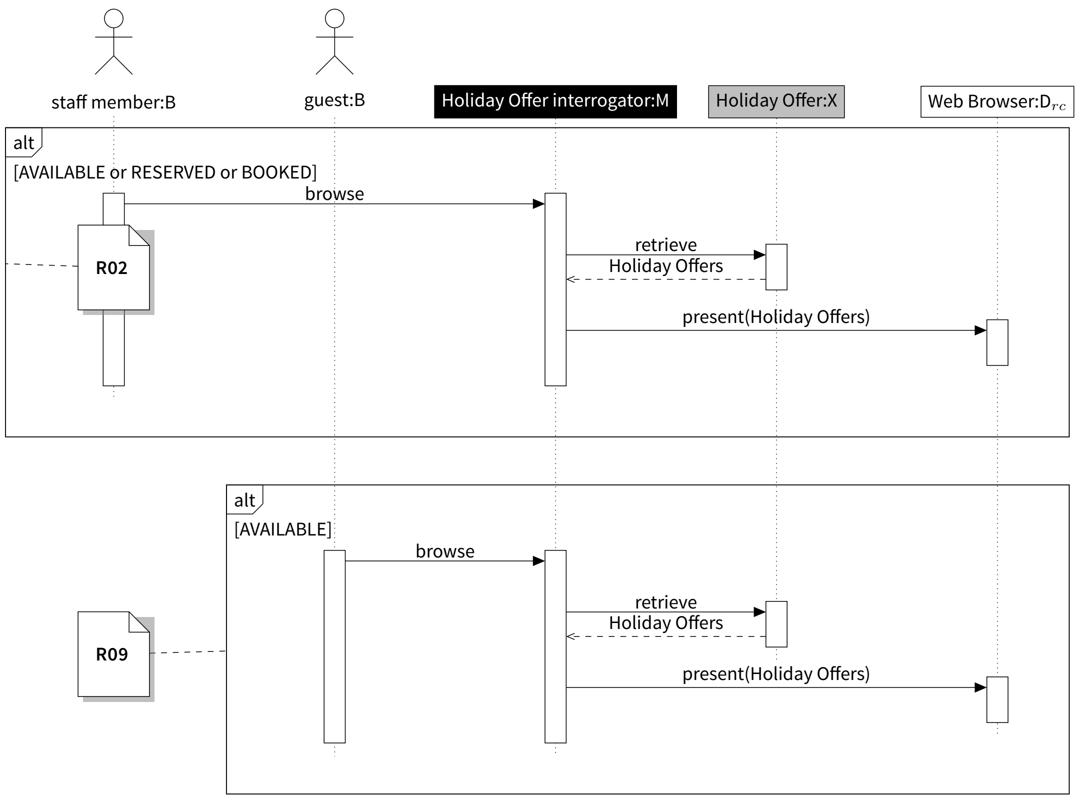


FIGURE 10.6 Specification of Requirements Work Package RWP02: Present Holiday Offers

10.4.3. Task scenarios of RWP03: Provide Invoice

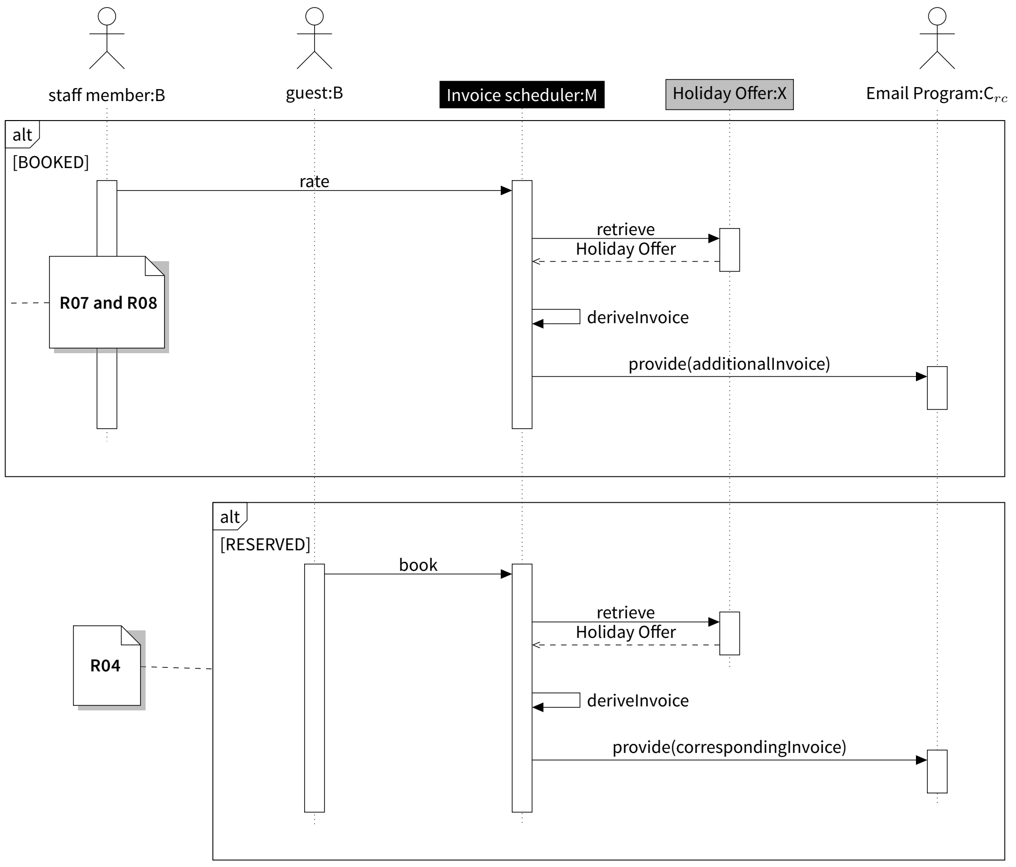


FIGURE 10.7 Specification of Requirements Work Package RWP03: Provide Invoice

10.5. Requirements Dependencies

10.5.1. Life-Cycle Expressions

According to [108, slide 489] the life-cycle expression composing all seven problems for the vacation rentals is:

- $LC_{guest} = (BrowseOfferings^{+}; [Book])^{*}$
  since booking a holiday offer should only be possible after browsing these.
- $LC_{staff\ member} = (Make \mid (BrowseBookings; [Pay|Rate]))^{*}$
- $LC_{vacation\ rentals} = (||_{i=1}^{n} LC_{guest_i}) \mid\mid (||_{j=1}^{m} LC_{staff\ member_j}) \mid\mid Reset^{*}$
  where $||_{i=1}^{n} LC_i$ denotes the parallel composition of n copies of life-cycle LC.

As already elaborated, the requirements decomposition for the vacation rentals changed due to the application of problem-based functional size measurement patterns. As a result there are now three requirements work packages RWP01 to RWP03 that allow for problem composition.

The changed life-cycle expression for the vacation rentals web application integrates these requirements work packages as follows:

- $LC'_{guest} = (RWP02.browse^{+}; [(RWP01.book \mid\mid RWP03.book)])^{*}$
- $LC'_{staff\ member} = (RWP01.make \mid (RWP02.browse; [RWP01.record \mid RWP03.rate]))^{*}$
- $LC'_{vacation\ rentals} = (||_{i=1}^{n} LC_{guest_i}) \mid\mid (||_{j=1}^{m} LC_{staff\ member_j})$

The dot-notation between the requirements work package name and one of its operations indicates to which variant of the measurable problem the life-cycle refers to. Seemingly, the difference between these two problem compositions for the vacation rentals consists in a simple problem renaming, but this does not tell the entire truth.

Comparing the life-cycle expressions for LC_{guest} with each other reveals that the former $Book$ problem involves a mixed composition of several kinds of problems.

First, each $book$ing results in a status change of an holiday offer via software functionality in RWP01 that represents an external input. In addition it involves sending an invoice to the respective guest via RWP03, which represents an external output. The life-cycle for LC'_{guest} reveals this inter-problem relationships of different measurable problems and therewith their respective requirements dependencies. Different problems are put to different requirements work packages and then composed by life-cycle expressions. That is why RWP01 and RWP03 substitute the former $Book$ problem in LC_{guest} and are composed in parallel with respect to the operation $book$ in LC'_{guest}.

Second, the $Reset$ problem, which is considered as an “internal operation” in the original requirements decomposition, is now reasonably mounted to the problem context of the “user-recognizable operation” $book$ in RWP01. This establishes intra-problem cohesiveness of dependent software functionality, that is implemented in one requirements work package. In addition, this contributes to the user view on requirements as requested by functional size measurement. Therefore, the $Reset$ problem is not a member of the $LC_{vacation\ rentals}$ anymore.

In addition, the life-cycles for LC_{guest} differ in respect to the former $BrowseOfferings$ problem, which is not separated from the $BrowseBookings$ problem anymore. Their problem scopes have such a huge overlap, that these two problems can be reasonably implemented in one requirements work package RWP02, following the criteria given on page 66. Clarifying requirements dependencies in this way, introduces a means for discovering reusable software functionality.

Comparing the life-cycle expressions for $LC_{staff\ member}$ with each other reveals that the former $Make$ problem covers more than the initialization concern for creating a holiday offer in $LC'_{staff\ member}$ and that the guest and staff member “usage protocols” of the vacation rentals have overlaps, which should not be ignored for several reasons.

In the first case, the former $Make$ and Pay problems become combined to one requirements work package RWP01, which covers all changes to the status of holiday offers.

In the second case, identifying overlaps in the problem composition contributes to the resolution of requirements conflicts and prioritization. The requirements work package RWP01 appears in the life-cycle expression for $LC'_{staff\ member}$ and LC'_{guest}, which indicates the importance of this set of software functionality to several user groups. This information is not for direct reading in the former life-cycle expressions.

Knowing occurrences and the precedence of a requirement work package in the context of a software’s life-cycle model supports an informed decision making in the project planning step as further detailed and utilized in part III Problem-Based Project Adaptation.

10.5.2. State Transition Diagram

Comparing the life-cycle expressions for $LC_{vacation\ rentals}$ with each other reveals that the final need of the vacation rentals web application to satisfy all life-cycle expressions in parallel does not change. $LC'_{staff\ member}$ and LC'_{guest} are synchronized with respect to the status of a holiday offer, which makes the states of the vacation rentals.

It also shows that by the use of requirements work packages as a means for problem composition only interaction between a user and the system are in the focus of considerations. The $Reset$ problem does not apply in this connection, since it is not directly triggered by a user.

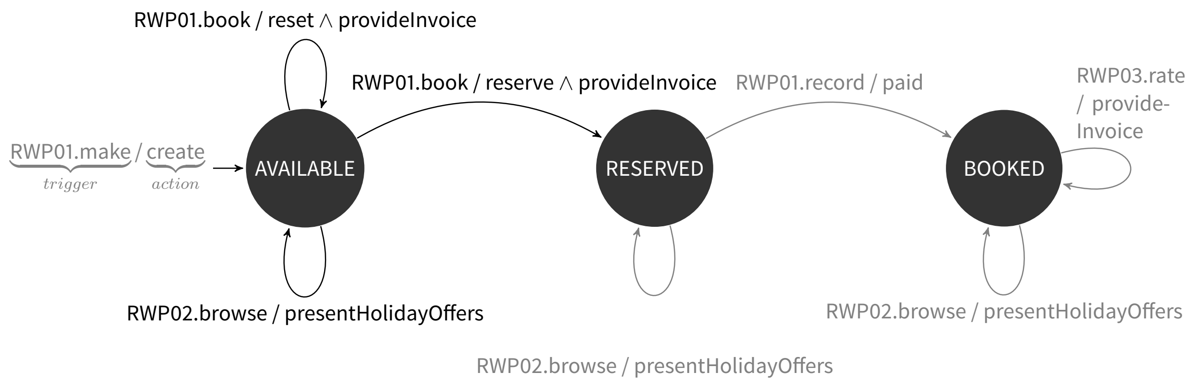

RWP01.book / reset ∧ provideInvoice
RWP01.book / reserve ∧ provideInvoice
RWP01.record / paid
RWP03.rate / provide-Invoice
RWP01.make / create
trigger
action
AVAILABLE
RESERVED
BOOKED
RWP02.browse / presentHolidayOffers
RWP02.browse / presentHolidayOffers
RWP02.browse / presentHolidayOffers

FIGURE 10.8 State machine as joint usage protocol for the vacation rentals web application

Figure 10.8 gives a state machine representation for $LC_{vacation\ rentals}$.

The $trigger$ of each transition is marked as $<$ $requirements\ work\ package$ $>$. $<$ $operation$ $>$ and coincides with a respective message to the machine domain in one alt-fragment of a requirements work package.

The $action$ taken when activating a transition is marked as $/$ $<$ $operation$ $>$ and coincides with a respective message to a constrained problem domain in the same alt-fragment of a requirements work package.

Regarding the occurrences of requirements work packages in the life-cycle for $LC_{vacation\ rentals}$: RWP01 is the one, whose software functionality is most used.

Regarding the precedence of requirements work packages in the life-cycle for $LC_{vacation\ rentals}$: RWP01 is central for initializing the vacation rentals services due to the $create$ action.

Figure 10.9 gives an alternative state machine for representing the dependencies of the vacation rentals's functional user requirements. Its transitions conform to the use cases given in figure 10.4 on page 209.

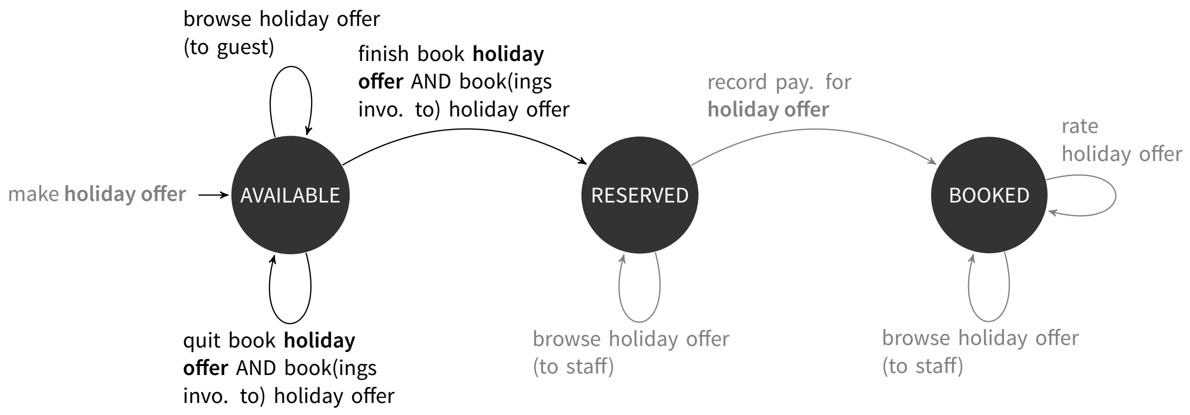

browse holiday offer (to guest)
finish book **holiday offer** AND book(ings invo. to) holiday offer
record pay. for **holiday offer**
rate holiday offer
make **holiday offer**
AVAILABLE
RESERVED
BOOKED
quit book **holiday offer** AND book(ings invo. to) holiday offer
browse holiday offer (to staff)
browse holiday offer (to staff)

FIGURE 10.9 State machine applying UML use cases for the Vacation Rentals Web Application

11. Student Recruitment Web Portal

The student recruitment web portal [116] is an online application, which can be accessed via the weblink: http://www.way2studying.de/application/. It is built to support high school leavers from abroad to begin their studies in North-Rhine Westphalia (NRW), Germany within the same year of their secondary school exit examination. This becomes possible due to modifications of paragraph 49 in the university law of NRW in 2013. The University of Duisburg-Essen has decided to open its International Studies in Engineering (ISE) programme for this type of applicants. The student recruitment web portal assists the respective application procedure for these programme.

11.1. Requirements Decomposition

A set of six requirement statements FUR #01 to FUR #06 has been re-engineered from the user interface of the student recruitment web portal. Its decomposition into six independent problems by means of problem frames is given in table 11.1.

No.	Requirements Statement
FUR #01	A candidate receives an eMail for granting access to the application procedure.
FUR #02	A candidate inserts application data on the web.
FUR #03	A candidate reviews application data on the web.
FUR #04	A candidate reviews application data as PDF file.
FUR #05	A candidate submits application files on the web.
FUR #06	A program admin reviews all data and files for a candidate as one printout.

TABLE 11.1 Requirements for a student recruitment web portal

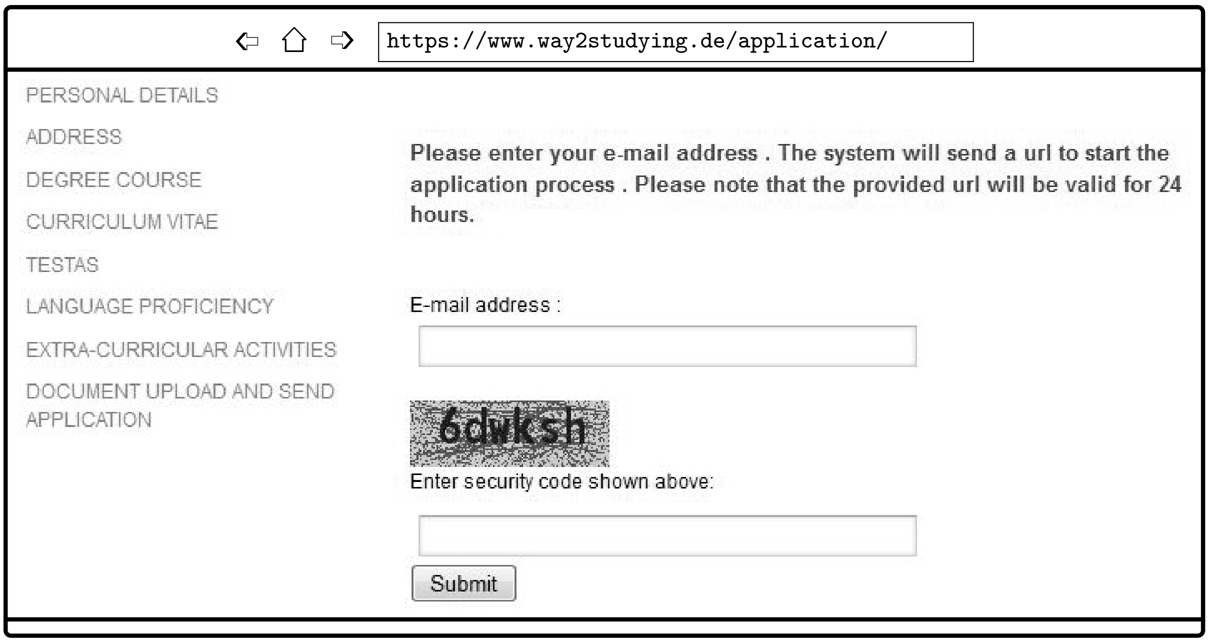
That each problem takes its own Requirements Work Package (RWP) as shown in table 11.2 is subject to chance in this specific application example.

No.	Problem Name	Problem Class: Problem Frame (FSM pattern)	RWP
FUR #01	grant access authorization	EO: commanded behaviour (#14)	RWP01
FUR #02	record candidate data	EI: simple workpieces (#02)	RWP02
FUR #03	review candidate data	EQ: query (#09)	RWP03
FUR #04	download candidate data	EO: commanded data-based control (#15)	RWP04
FUR #05	upload candidate files	EI: commanded model building (#05)	RWP05
FUR #06	compile candidate résumé	EQ: query (#09)	RWP06

TABLE 11.2 Decomposition of requirements for a student recruitment web portal

11.1.1. Problem description for FUR #01: Grant Access Authorization

Figure 11.1 shows a screenshot of the student recruitment web portal's landing page on accessing it via http://www.way2studying.de/application/. It is structured to a sidebar on the left, and a content page to the right. The sidebar provides means for navigating the website, the content page represents the user interface to give information or to get information from the website. That way, an applicant or respective candidate is supported in filling required application forms one after the other, and to submit necessary documents online.



https://www.way2studying.de/application/

PERSONAL DETAILS
ADDRESS
DEGREE COURSE
CURRICULUM VITAE
TESTAS
LANGUAGE PROFICIENCY
EXTRA-CURRICULAR ACTIVITIES
DOCUMENT UPLOAD AND SEND APPLICATION

Please enter your e-mail address . The system will send a url to start the application process . Please note that the provided url will be valid for 24 hours.

E-mail address :

6dwksh

Enter security code shown above:

Submit

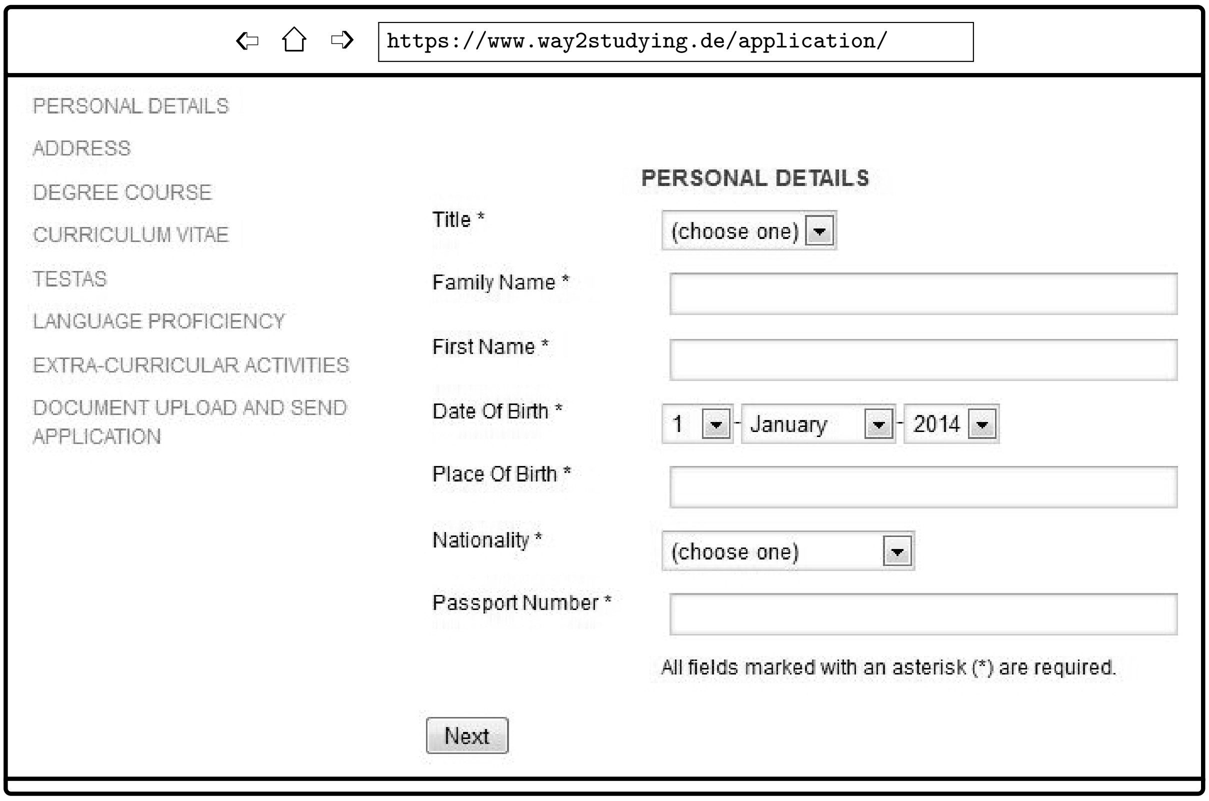
FIGURE 11.1 Screenshot of user interface covering FUR #01: Grant Access Authorization

The problem covered by FUR #01 can be described as follows: A valid *email address* and a correct *security code* must be provided by the candidate, in order to successfully request access to the application services of this website. After these data have been submitted, the website creates a link to the personal application page for this candidate, and sends the respective URL to the provided email address. This link is valid for twenty-four hours within the candidate can start the application process.

Figure 11.6 gives the respective requirements work package. It is an instance of the functional size measurement pattern #14 commanded behavior in table 5.5, where the candidate is a biddable and the email program is a causal problem domain. Based on this classification, the requirements work package's software functionality represents a measurable problem, whose constituent parts are sized according to the rules of an external output (EO).

11.1.2. Problem description for FUR #02: Record Candidate Data

Figure 11.2 shows a screenshot of the student recruitment web portal after the candidate activates the access link provided to her by email, see descriptions of FUR #01 on page 219. The application process starts by requesting the candidate to fill out several, subsequent forms with personal information. As can be followed by the sidebar, there is a sequence of seven forms, i.e. *personal details* to *extra curricular activities* to be completed by the candidate.


https://www.way2studying.de/application/

PERSONAL DETAILS
ADDRESS
DEGREE COURSE
CURRICULUM VITAE
TESTAS
LANGUAGE PROFICIENCY
EXTRA-CURRICULAR ACTIVITIES
DOCUMENT UPLOAD AND SEND APPLICATION

PERSONAL DETAILS

Title *	(choose one)
Family Name *	
First Name *	
Date Of Birth *	1 - January - 2014
Place Of Birth *	
Nationality *	(choose one)
Passport Number *	

All fields marked with an asterisk (*) are required.

FIGURE 11.2 Screenshot of user interface covering FUR #02: Record Candidate Data

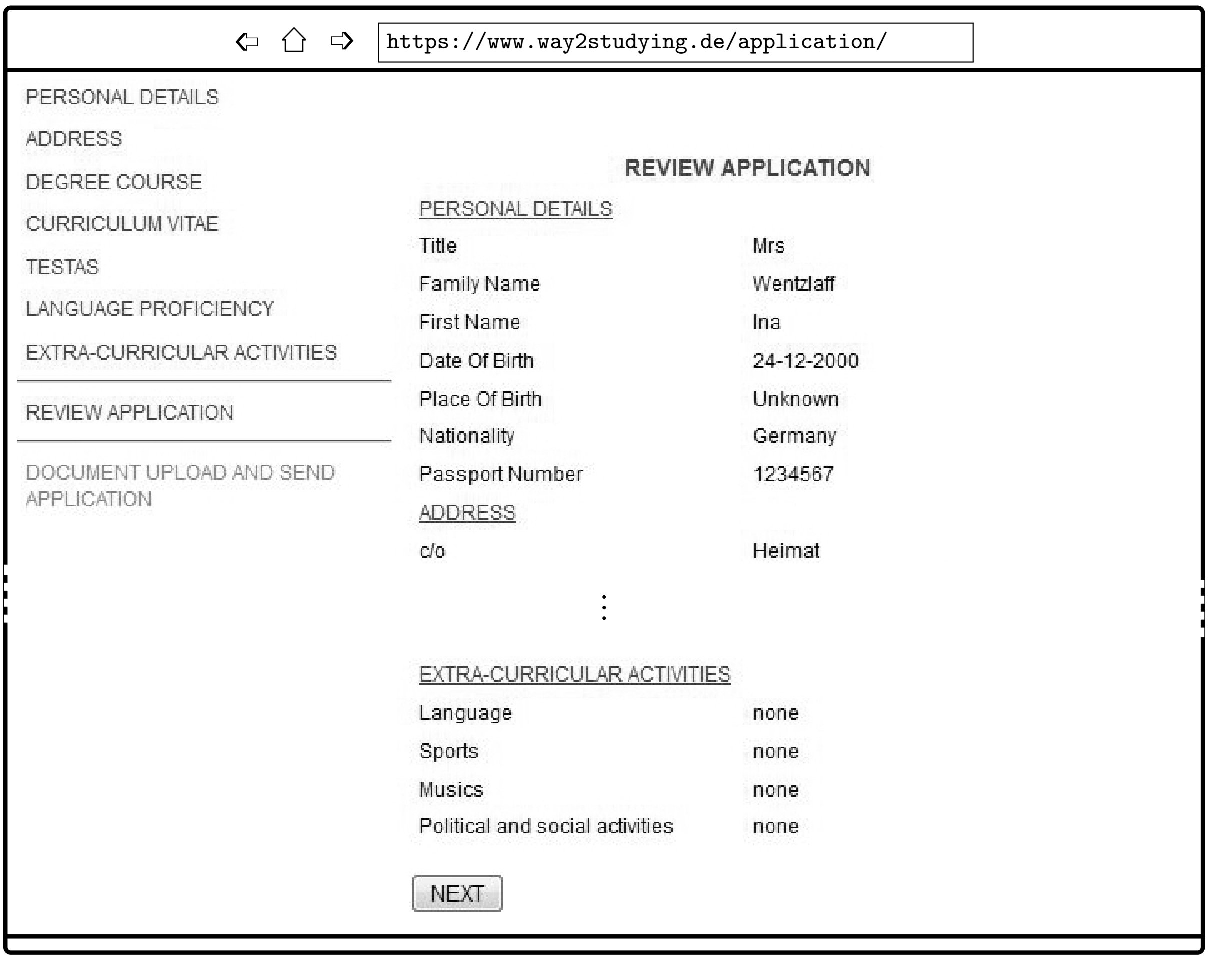
The problem covered by FUR #02 can be described as follows: The candidate fills the forms with personal information, that is stored and required for conducting the online application service.

Figure 11.7 gives the respective requirements work package. It is an instance of the functional size measurement pattern #02 simple workpieces in table 5.5, where the candidate is a biddable and the candidate data is a lexical problem domain. The shared phenomenon *FormData1..40* is a placeholder for the forty data fields, e.g. *title*, *family name*, *first name*, etc. that need to be completed by the candidate, and which are distributed over the above mentioned seven, subsequent forms.

Due to this classification by means of functional size measurement patterns, the requirements work package's software functionality represents a measurable problem, whose constituent parts are sized according to the rules of an external input (EI).

11.1.3. Problem description for FUR #03: Review Candidate Data

Figure 11.3 shows a snippet of a screenshot, that gives the student recruitment web portal after the candidate has completed all forms. Only then the link *review application* becomes available at the sidebar. The respective content page shows all the data provided by the candidate for applying to an ISE study, and which is stored to the data base of the student recruitment web portal.


https://www.way2studying.de/application/

REVIEW APPLICATION

PERSONAL DETAILS

Title	Mrs
Family Name	Wentzlaff
First Name	Ina
Date Of Birth	24-12-2000
Place Of Birth	Unknown
Nationality	Germany
Passport Number	1234567

ADDRESS

c/o	Heimat

⋮

EXTRA-CURRICULAR ACTIVITIES

Language	none
Sports	none
Musics	none
Political and social activities	none

FIGURE 11.3 Screenshot of user interface covering FUR #03: Review Candidate Data

The problem covered by FUR #03 can be described as follows: The candidate can review all provided, personal information that is stored to conduct the online application service.

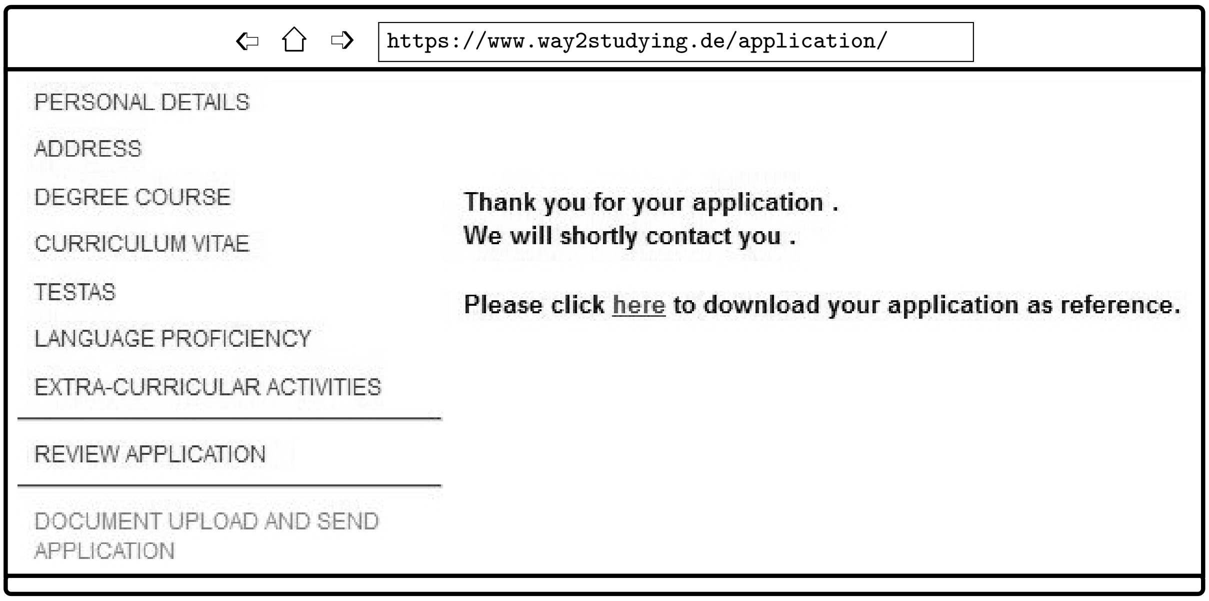
Figure 11.8 gives the respective requirements work package. It is an instance of the functional size measurement pattern #09 query in table 5.5, where the candidate is a biddable, the candidate data is a lexical problem domain, and the content page is shown by means of the web browser, which is a display domain.

The stored personal information represented by the shared phenomenon $FormData1..40$ is simply retrieved from the database and presented at the content page to the candidate.

Due to this classification by means of functional size measurement patterns, the requirements work package's software functionality represents a measurable problem, whose constituent parts are sized according to the rules of an external inquiry (EQ).

11.1.4. Problem description for FUR #04: Download Candidate Data

Figure 11.4 shows a screenshot of the student recruitment web portal, which becomes available after the entire application procedure is completed and all personal information and files provided by the candidate are recorded.


https://www.way2studying.de/application/
PERSONAL DETAILS
ADDRESS
DEGREE COURSE
CURRICULUM VITAE
TESTAS
LANGUAGE PROFICIENCY
EXTRA-CURRICULAR ACTIVITIES
REVIEW APPLICATION
DOCUMENT UPLOAD AND SEND APPLICATION

Thank you for your application .
We will shortly contact you .

Please click here to download your application as reference.

FIGURE 11.4 Screenshot of user interface covering FUR #04: Download Candidate Data

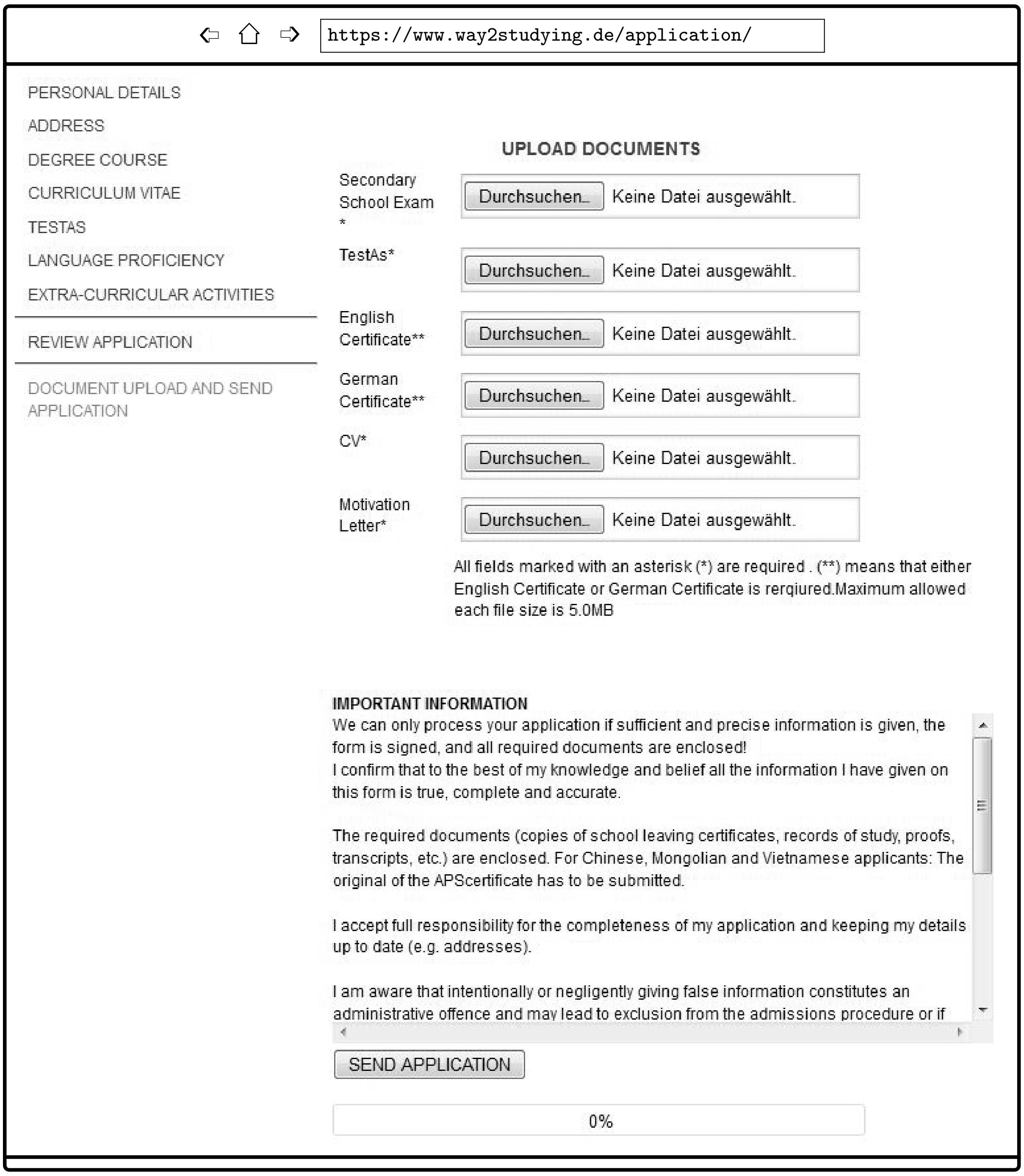
The problem covered by FUR #04 can be described as follows: The candidate can request an overview given as a PDF file, which comprises all personal information, that has been provided to the online application service by filling in the forms.

Figure 11.9 gives the respective requirements work package. It is an instance of the functional size measurement pattern #15 commanded data-based control in table 5.5, where the candidate is a biddable, the candidate data is a lexical problem domain, and the PDF viewer is an other software application given as a causal domain, that is used to read the respective summary file, that is to produce by the student recruitment web portal. Please note, that uploaded candidate files such as a curriculum vitae or a certification, are not integrated to this PDF file.

Due to this classification by means of functional size measurement patterns, the requirements work package's software functionality represents a measurable problem, whose constituent parts are sized according to the rules of an external output (EO).

11.1.5. Problem description for FUR #05: Upload Candidate Files

Figure 11.5 shows a screenshot of the student recruitment web portal that relates to the sidebar link *document upload and send application*. It gives the final content page, which enables the candidate to upload additional documents and to complete the application.



https://www.way2studying.de/application/

PERSONAL DETAILS
ADDRESS
DEGREE COURSE
CURRICULUM VITAE
TESTAS
LANGUAGE PROFICIENCY
EXTRA-CURRICULAR ACTIVITIES
REVIEW APPLICATION
DOCUMENT UPLOAD AND SEND APPLICATION

UPLOAD DOCUMENTS

Secondary School Exam *	Durchsuchen... Keine Datei ausgewählt.
TestAs*	Durchsuchen... Keine Datei ausgewählt.
English Certificate**	Durchsuchen... Keine Datei ausgewählt.
German Certificate**	Durchsuchen... Keine Datei ausgewählt.
CV*	Durchsuchen... Keine Datei ausgewählt.
Motivation Letter*	Durchsuchen... Keine Datei ausgewählt.

All fields marked with an asterisk (*) are required . (**) means that either English Certificate or German Certificate is rerqiured.Maximum allowed each file size is 5.0MB

IMPORTANT INFORMATION
We can only process your application if sufficient and precise information is given, the form is signed, and all required documents are enclosed!
I confirm that to the best of my knowledge and belief all the information I have given on this form is true, complete and accurate.

The required documents (copies of school leaving certificates, records of study, proofs, transcripts, etc.) are enclosed. For Chinese, Mongolian and Vietnamese applicants: The original of the APScertificate has to be submitted.

I accept full responsibility for the completeness of my application and keeping my details up to date (e.g. addresses).

I am aware that intentionally or negligently giving false information constitutes an administrative offence and may lead to exclusion from the admissions procedure or if

SEND APPLICATION

0%

FIGURE 11.5 Screenshot of user interface covering FUR #05: Upload Candidate Files

The problem covered by FUR #05 can be described as follows: The candidate can upload additional documents to complete and finally submit the application.

Figure 11.10 gives the respective requirements work package. It is an instance of the functional size measurement pattern #05 commanded model building in table 5.5, where the candidate is a biddable, the candidate files are a lexical problem domain, and the file manager given as a causal domain is an other software application, which allows the candidate to select the files for the upload. These files are represented by the shared phenomena *CF1* to *CF6*.

Due to this classification by means of functional size measurement patterns, the requirements work package's software functionality represents a measurable problem, whose constituent parts are sized according to the rules of an external input (EI).

11.1.6. Problem description for FUR #06: Compile Candidate Résumé

There is no screenshot for FUR #06 of the student recruitment web portal, since for this requirement, there is only a call at the command line available, named “admin interface” in the following. Nevertheless, it is a user interface to the application, and considered next.

The problem covered by FUR #06 can be described as follows: The admin can compile each application’s data and files to one printout for each candidate. This printout is checked in the admission process. The “admin interface” is invoked after the application period is closed.

Figure 11.11 gives the respective requirements work package. It is an instance of the functional size measurement pattern #09 query in table 5.5, where the admin is a biddable, the candidate data and files are lexical problem domains, and the printout of each application is managed by a printer device, which is classified as a display domain.

Due to this classification by means of functional size measurement patterns, the requirements work package’s software functionality represents a measurable problem, whose constituent parts are sized according to the rules of an external inquiry (EQ).

11.2. Requirements Measurement

The previous sections on page 218ff give all the details on the setup of each requirements work package for the student recruitment web portal. This section is concerned with the sample application of the counting procedure as introduced in section 6.3 on page 81 to these measurable problems implemented in FUR #01 to FUR #06.

Prio[LC]	Product Backlog Item	Basic FSM Pattern[PB]	EP	FP ▲
3.	FUR #05 Upload Candidate Files	#05 commanded model building	EI	18
2.	FUR #02 Record Candidate Data	#02 simple workpieces	EI	13
1.	FUR #01 Grant Access Authorization	#14 commanded behavior	EO	17
5.	FUR #04 Download Candidate Data	#15 commanded data-based control	EO	17
5.	FUR #03 Review Candidate Data	#09 query	EQ	16
4.	FUR #06 Compile Candidate Résumé	#09 query	EQ	16

[LC] Priority of requirements work packages with respect to life-cycle given on page 167
[PB] Problem-Based Functional Size Measurement Patterns according to table 5.5 on page 72
Elementary Process (EP), Function Points (FP)

TABLE 11.3 Augmented Product Backlog of a student recruitment web portal

Table 11.3 summaries the results of problem-based estimating as applied to the requirements work packages for the student recruitment web portal. For the sake of readability, the product backlog items are ordered according to their determined function points and related elementary processes. The priority of each item depends on their occurrence within the life-cycle expressions for the student recruitment web portal as discussed on page 167.

The following sections comment on the results determined for each requirements work package, where the details of each count are given on pages 228ff. Please reconsider that this work strives towards consistent point values for recognizable requirements. Therefore, it follows the trend to tailor a given functional size measurement method instead of introducing a new one [25, page 180]. It attends to size right by application of the ISO 20926 standard, rather than to question if the ISO 20926 standard applies a right size.

Requirements work packages FUR #02 and FUR #05 implement a measurable problem, which relates to an external input. According to criteria **UF.C1** given on page 66, they cannot be combined to one unique set of software functionality, since they do not share the same constrained problem domain.

While the absolute number of data element types that relate to data functions in FUR#02 ($ILF_{DET} + EIF_{DET} = 40\ DET$) is greater than in FUR#05 ($ILF_{DET} + EIF_{DET} = 12\ DET$), the requirements work package size for $FUR\#02_{size} = 13\ FP$ is smaller than for $FUR\#05_{size} = 18\ FP$. This is due to the same data function complexity class for these DET values, which is in both cases low with respect to column one and two in [117, table A.1, page 23] given on page 262 in the appendix.

The functional size of requirements work packages FUR #02 and FUR #05 is determined according to the rules of an external input, but FUR #05 involves more data movements. It comprises more file types referenced ($TF_{FTR} = 3$) with relevant information, i.e. problem domains with symbolic phenomena than FUR #02 ($TF_{FTR} = 2$), which must be processed.

What can be seen here is that according to ISO 20926, the number of interfaces that carry some relevant information counts more than the absolute number of information items implemented in one respective requirements work package.

Requirements work packages FUR #01 and FUR #04 implement a measurable problem, which relates to an external output. According to criteria **UF.C1** given on page 66, they cannot be combined to one unique set of software functionality, since they do not share the same constrained problem domain.

While the absolute number of data element types related to data functions in FUR#01 ($ILF_{DET}+EIF_{DET} = 3\,DET$) is significantly smaller than in FUR#04 ($ILF_{DET}+EIF_{DET} = 40\,DET$), the requirements work package size for $FUR\#01_{size} = 17\,FP$ is the same as for $FUR\#04_{size} = 17\,FP$.

Both measurable problems comprise the same number of file types referenced ($TF_{FTR} = 2$), i.e. problem domains with relevant information, which must be processed. However, FUR #01 involves more information that crosses the application boundary. It receives data from the candidate, which is then processed and forwarded via email. In contrast, FUR #04 simply compiles data from an internal logical file to a PDF document.

As given by the requirement work packages at hand, it is illustrated that according to ISO 20926 processing of information that is not user-recognizable counts less than processing of information, which crosses the application boundary. In this regard, it is additionally shown that the counting philosophy of ISO 20926 is met by the problem-based functional size measurement executed for FUR#01 and FUR#04.

Requirements work packages FUR #03 and FUR #06 implement a measurable problem, which relates to an external inquiry. Both product backlog items are instances of a $query$ frame, but anyway they cannot be combined to one unique set of software functionality. Criteria **UF.C1** given on page 66 is not fulfilled, since they do not share the same constrained problem domain.

Both measurable problems comprise the same total number of data element types related to their data functions $ILF_{DET}+EIF_{DET} = 40\,DET$, and they process a comparable number of data that crosses the application boundary, which is $TF_{DET} = 42$ in case of FUR #03 and $TF_{DET} = 48$ in case FUR #06.

What can be seen by these requirements work packages is that these are equally filled with relevant information that counts in their functional size measurement according to ISO 20926. The need of FUR #06 to access more internal logical files ($3\,ILF$) than FUR #03 ($2\,ILF$) is subordinate to the total number of data element types that these ILF hold, when determining their functional size. This contributes to the measurement practice, that internal logical files models data with respect to their logical dependence rather than to their physical allocation.

FUR #03 and FUR #06 are instances of the same measurable problem class and equally filled with information relevant for determining their functional size. That the sizes of these two requirements work packages conform to each other is an example for the capability of problem-based estimating estimating to produce consistent function point values for comparable sets of requirements.

11.2.1. Problem count of RWP for FUR #01: Grant Access Authorization

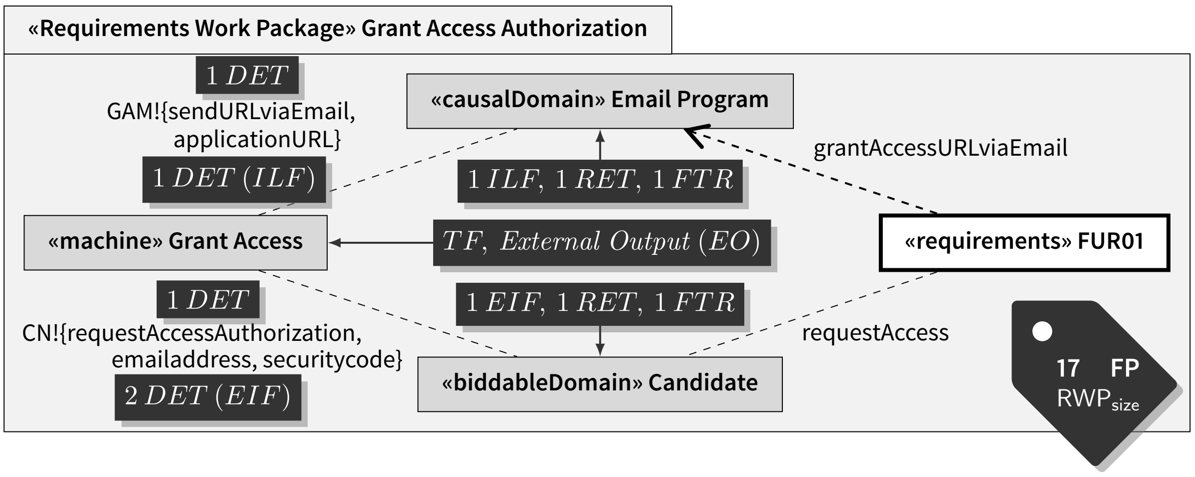

«Requirements Work Package» Grant Access Authorization
1 DET
GAM!{sendURLviaEmail, applicationURL}
1 DET (ILF)
«causalDomain» Email Program
1 ILF, 1 RET, 1 FTR
grantAccessURLviaEmail
«machine» Grant Access
TF, External Output (EO)
«requirements» FUR01
1 DET
CN!{requestAccessAuthorization, emailaddress, securitycode}
2 DET (EIF)
1 EIF, 1 RET, 1 FTR
requestAccess
«biddableDomain» Candidate
17 FP
RWPsize

FIGURE 11.6 Counting Requirements Work Package FUR #01: Grant Access Authorization

Comments on counting process activity	Results of activity
1. Classify FUR by Functional Size Measurement Patterns.	
FUR #01: Grant Access Authorization is a measurable problem, because it fits the *commanded behavior* FSM pattern #14 in table 5.5. Applied validation conditions: V.i - V.iii	commanded behavior problem
2. Determine Data Functions.	
FUR #01: Grant Access Authorization has two problem domains, namely the causal domain *Email Program* and the biddable domain *Candidate*.	
2.a Identify problem domains as data functions.	
The domain *Email Program* shares one symbolic phenomenon *applicationURL* with the machine domain. The domain *Email Program* can be classified as data function. The domain *Candidate* shares two symbolic phenomena at the machine interface, namely *emailaddress* and *securitycode*. It is a data function, too. Applied validation conditions: V.iv	2 data functions: Email program and Candidate
2.b Classify data functions into ILF or EIF.	
This measurable problem has two data functions, of which *Email Program* is an internal logical file, and *Candidate* is an external interface file. Applied validation conditions: V.vi, V.vii, (V.viii,) V.ix	1 ILF: Email Program 1 EIF: Candidate
2.c Count DET for each data function.	
The data function *Email Program* shares one symbolic phenomenon at the machine interface, namely *applicationURL*, which represents the only data element type of this data function.	$DET_{email\ program} = 1$
The data function *Candidate* shares two symbolic phenomenon at the machine interface, namely *emailaddress* and *securitycode*, which represent the data element types of this data function. Applied validation conditions: V.xi	$DET_{candidate} = 2$
2.d Count RET for each data function.	
There are two data functions *Email Program* and *Candidate*, each represents 1 RET. Applied validation conditions: V.xii	$RET_{email\ program} = 1$ $RET_{candidate} = 1$
2.e Determine functional complexity for data functions.	

Comments on counting process activity	Results of activity
$Email\ Program$ is the one data function, i.e. an ILF in this measurable problem, which has 1 DET according to step 2.c and represents 1 RET according to step 2.d. Respectively, $ILF_{DET} = \sum_{i=1}^{n} DET_{ILF_i} = DET_{email\ program}$ and $ILF_{RET} = \sum_{i=1}^{n} RET_{ILF_i} = RET_{email\ program}$.	$ILF_{DET} = 1$ $ILF_{RET} = 1$
Table A.1 of ISO 20926 given in the appendix on page 262 is used to determine the respective data function complexity $ILF_{Complexity}(ILF_{RET}, ILF_{DET})$ by means of these RET and DET values, that is $ILF_{Complexity}(1, 1) = low$.	$ILF_{Complexity} = low$
$Candidate$ is the other data function, i.e. an EIF in this measurable problem, which has 2 DET according to step 2.c and represents 1 RET according to step 2.d. Respectively, $EIF_{DET} = \sum_{i=1}^{n} DET_{EIF_i} = DET_{candidate}$ and $EIF_{RET} = \sum_{i=1}^{n} RET_{EIF_i} = RET_{candidate}$.	$EIF_{DET} = 2$ $ILF_{RET} = 1$
Table A.1 of ISO 20926 given in the appendix on page 262 is used to determine the respective data function complexity $EIF_{Complexity}(EIF_{RET}, EIF_{DET})$ by means of these RET and DET values, that is $EIF_{Complexity}(1, 2) = low$. Applied validation conditions: V.xiii, V.xiv, V.xvii, V.xx, V.xxi	$EIF_{Complexity} = low$
2.f Determine functional size for data functions.	
Table A.2 of ISO 20926 given in the appendix on page 262 is applied to determine the respective data function size $ILF_{Size}(ILF_{Complexity}, ILF)$ using its data function complexity determined in the previous step 2.e., that is $ILF_{Size}(low, ILF) = 7$. The data function size $EIF_{Size}(EIF_{Complexity}, EIF)$ is determined respectively $EIF_{Size}(low, EIF) = 5$. Applied validation conditions: V.xxiii, V.xxiv	ILF_{Size} = 7 function points EIF_{Size} = 5 function points
3. Determine Transactional Function.	
FUR #01: Grant Access Authorization has one machine domain $Grant\ Access$.	
3.a Identify machine domain as transactional function.	
The machine domain $Grant\ Access$ represents the transactional function in this measurable problem.	transactional function: grant access
3.b Classify transactional function as either EI, EQ, or EO.	
which is a functional size measurement pattern as defined in table 5.5 for determining the functional size of an external output (EO). Applied validation conditions: V.xxv	$TF_{type} = EO$
3.c Count FTR for transactional function.	
$Grant\ Access$ involves two machine interfaces to a data function as defined in step 2.a, which are the file type referenced to consider in this step. One data function $email\ program$ and one external interface file $candidate$ results in $n = 1$ and $m = 1$ for TF_{FTR} = n ILF + m EIF = 1 + 1 = 2. Applied validation conditions: V.xxvi	$TF_{FTR} = 2$
3.d Count DET for transactional function.	
At the machine interface of $Grant\ Access$ are two symbolic phenomena and one causal shared with $Candidate$, and the machine shares one causal and one symbolic phenomenon with the $Email\ Program$. Each of these five phenomena crosses the application boundary and thus count in the transactional function as DET. Applied validation conditions: V.xxvii, V.xxix, V.xxx, V.xxxi	$TF_{DET} = 5$
3.e Determine functional complexity for transactional function.	
Table A.4 for EO of ISO 20926 given in the appendix on page 262 is used to determine the transactional function complexity $TF_{Complexity}(TF_{Type}, TF_{FTR}, TF_{DET}) = TF_{Complexity}(EO, 2, 5)$ of $Grant\ Access$ by means of the FTR and DET values from step 3.c and 3.d. Applied validation conditions: V.xxxii V.xxxii	$TF_{Complexity} = average$
3.f Determine functional size for transactional function.	

Comments on counting process activity	Results of activity
Table A.5 of ISO 20926 given in the appendix on page 262 is applied to determine the respective transactional function size of *Grant Access* $TF_{Size}(TF_{Complexity}, TF_{Type}) = TF_{Size}(average, EO)$ by using its transactional function complexity determined in the previous step 3.e. Applied validation conditions: V.xxxiii	$TF_{Size} =$ 5 function points
4. Report Functional Size for FUR.	
by $MeasurableProblem_{size} = ILF_{Size} + EIF_{Size} + TF_{Size} = 7 + 5 + 5.$ Applied validation conditions: V.xxxiv	$FUR\#01_{size}$ **= 17 function points**

11.2.2. Problem count of RWP for FUR #02: Record Candidate Data

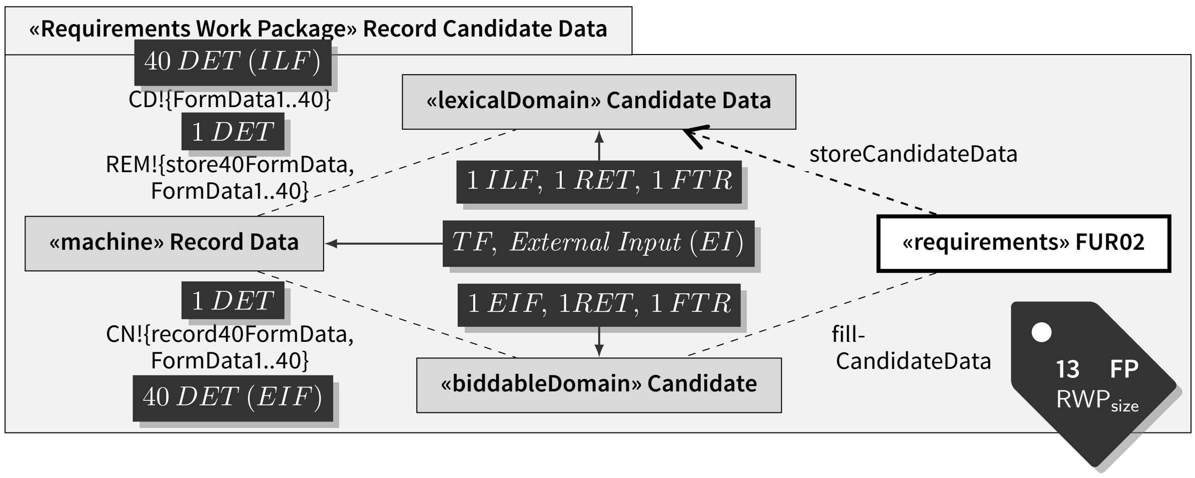


FIGURE 11.7 Counting Requirements Work Package FUR #02: Record Candidate Data

Comments on counting process activity	Results of activity
1. Classify FUR by Functional Size Measurement Patterns.	
Applied validation conditions: V.i - V.iii	simple workpieces problem

Comments on counting process activity	Results of activity
2. Determine Data Functions.	
2.a Identify problem domains as data functions.	
Applied validation conditions: V.iv	2 data functions: Candidate and Candidate Data
2.b Classify data functions into ILF or EIF.	
Applied validation conditions: V.vi, V.vii, (V.viii,) V.ix	1 EIF: Candidate
	1 ILF: Candidate Data
2.c Count DET for each data function.	
Count symbolic phenomena	$DET_{candidate} = 40$
Applied validation conditions: V.xi	$DET_{candidate\ data} = 40$
2.d Count RET for each data function.	
Applied validation conditions: V.xii	$RET_{candidate} = 1$
	$RET_{candidate\ data} = 1$
2.e Determine functional complexity for data functions.	
Only one ILF $ILF_{DET} = \sum_{i=1}^{n} DET_{ILF_i} = DET_{candidate\ data}$ and	$ILF_{DET} = 40$
$ILF_{RET} = \sum_{i=1}^{n} RET_{ILF_i} = RET_{candidate\ data}$.	$ILF_{RET} = 1$
Only one EIF $EIF_{DET} = \sum_{i=1}^{n} DET_{EIF_i} = DET_{candidate}$ and $EIF_{RET} =$	~~$EIF_{DET} = 40$~~
$\sum_{i=1}^{n} RET_{EIF_i} = RET_{candidate}$.	$EIF_{RET} = 1$
But $EIF_{DET} - k$, and $k = 40$ equal phenomena	$EIF_{DET} = 0$
$ILF_{Complexity}(ILF_{RET}, ILF_{DET}) = ILF_{Complexity}(1, 40)$	$ILF_{Complexity} = low$
$EIF_{Complexity}(EIF_{RET}, EIF_{DET}) = EIF_{Complexity}(1, 0)$	$EIF_{Complexity} = \{n/a\}$
Applied validation conditions: V.xiii, V.xiv, V.xvii, V.xx, V.xxi	
2.f Determine functional size for data functions.	
$ILF_{Size}(ILF_{Complexity}, ILF) = ILF_{Size}(low, ILF) = 7$	ILF_{Size} = 7 function points
$EIF_{Size}(EIF_{Complexity}, EIF) = EIF_{Size}(\{n/a\}, EIF) = 0$	EIF_{Size} = 0 function points
Applied validation conditions: V.xxiii, V.xxiv	
3. Determine Transactional Function.	
3.a Identify machine domain as transactional function.	
	transactional function: record data
3.b Classify transactional function as either EI, EQ, or EO.	
simple workpieces	$TF_{type} = EI$
Applied validation conditions: V.xxv	
3.c Count FTR for transactional function.	
TF_{FTR} = n ILF + m EIF = 1 + 1 = 2.	$TF_{FTR} = 2$
Applied validation conditions: V.xxvi	
3.d Count DET for transactional function.	
Phenomena at machine interface to lexical domain do not count, that is only candidate phenomena are counted	$TF_{DET} = 21$
Applied validation conditions: V.xxvii, V.xxix, V.xxx, V.xxxi	
3.e Determine functional complexity for transactional function.	
$TF_{Complexity}(TF_{Type}, TF_{FTR}, TF_{DET}) = TF_{Complexity}(EI, 2, 21)$	$TF_{Complexity} = high$
Applied validation conditions: V.xxxii	
3.f Determine functional size for transactional function.	
$TF_{Size}(TF_{Complexity}, TF_{Type}) = TF_{Size}(high, EI)$	TF_{Size} = 6 function points
Applied validation conditions: V.xxxiii	
4. Report Functional Size for FUR.	
$MeasurableProblem_{size} = ILF_{Size} + EIF_{Size} + TF_{Size} = 7 + 0 + 6$	$FUR\,\#02_{size} =$
Applied validation conditions: V.xxxiv	**13 function points**

11.2.3. Problem count of RWP for FUR #03: Review Candidate Data

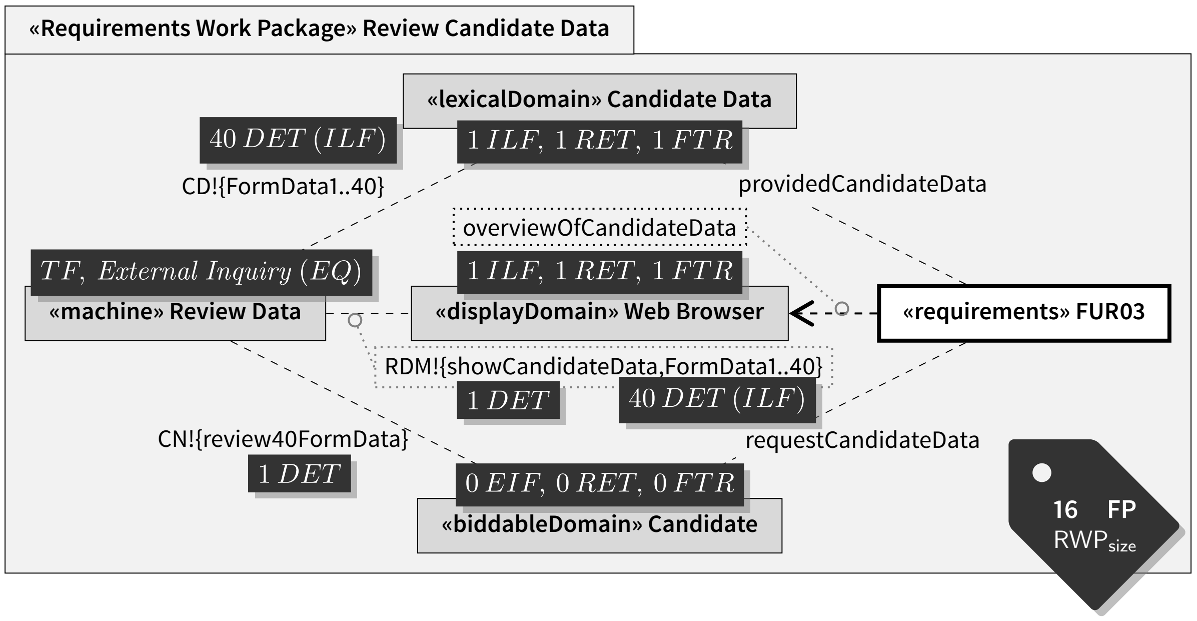


FIGURE 11.8 Counting Requirements Work Package FUR #03: Review Candidate Data

Comments on counting process activity	Results of activity
1. Classify FUR by Functional Size Measurement Patterns.	
Applied validation conditions: V.i - V.iii	query problem
2. Determine Data Functions.	
2.a Identify problem domains as data functions.	
Applied validation conditions: V.iv	2 data functions: Web Browser and Candidate Data
2.b Classify data functions into ILF or EIF.	
Applied validation conditions: V.vi, V.vii, V.x	1 ILF: Web Browser 1 ILF: Candidate Data
2.c Count DET for each data function.	
Count symbolic phenomena Applied validation conditions: V.xi	$DET_{web\ browser} = 40$ $DET_{candidate\ data} = 40$
2.d Count RET for each data function.	
Applied validation conditions: V.xii	$RET_{web\ browser} = 1$ $RET_{candidate\ data} = 1$
2.e Determine functional complexity for data functions.	
There are two ILF $ILF_{DET} = \sum_{i=1}^{n} DET_{ILF_i} = DET_{web\ browser} + DET_{candidate\ data}$ and $ILF_{RET} = \sum_{i=1}^{n} RET_{ILF_i} = RET_{web\ browser} + RET_{candidate\ data}$. No EIF.	~~$ILF_{DET} = 80$~~ $ILF_{RET} = 2$
But $ILF_{DET} - k$, and $k = 40$ equal phenomena $ILF_{Complexity}(ILF_{RET}, ILF_{DET}) = ILF_{Complexity}(2, 40)$ $EIF_{Complexity}(EIF_{RET}, EIF_{DET}) = EIF_{Complexity}(0, 0)$ Applied validation conditions: V.xiii, V.xvi, V.xvii, V.xix, V.xx	$ILF_{DET} = 40$ $ILF_{Complexity} = average$ $EIF_{Complexity} = \{n/a\}$
2.f Determine functional size for data functions.	
$ILF_{Size}(ILF_{Complexity}, ILF) = ILF_{Size}(average, ILF) = 10$ $EIF_{Size}(EIF_{Complexity}, EIF) = EIF_{Size}(\{n/a\}, EIF) = 0$ Applied validation conditions: V.xxii, V.xxiii	$ILF_{Size} = 10$ function points $EIF_{Size} = 0$ function points

Comments on counting process activity	Results of activity
3. Determine Transactional Function. 3.a Identify machine domain as transactional function.	
	transactional function: Review Data
3.b Classify transactional function as either EI, EQ, or EO.	
query Applied validation conditions: V.xxv	$TF_{type} = EQ$
3.c Count FTR for transactional function.	
TF_{FTR} = n ILF + m EIF = 2 + 0 = 2. Applied validation conditions: V.xxvi	$TF_{FTR} = 2$
3.d Count DET for transactional function.	
Only candidate and web browser phenomena are counted Applied validation conditions: V.xxix, V.xxx, V.xxxi	$TF_{DET} = 42$
3.e Determine functional complexity for transactional function.	
$TF_{Complexity}(TF_{Type}, TF_{FTR}, TF_{DET}) = TF_{Complexity}(EQ, 2, 42)$ Applied validation conditions: V.xxxii	$TF_{Complexity} = high$
3.f Determine functional size for transactional function.	
$TF_{Size}(TF_{Complexity}, TF_{Type}) = TF_{Size}(high, EQ)$ Applied validation conditions: V.xxxiii	TF_{Size} = 6 function points
4. Report Functional Size for FUR.	
$MeasurableProblem_{size} = ILF_{Size} + EIF_{Size} + TF_{Size} = 10 + 0 + 6$ Applied validation conditions: V.xxxiv	$FUR\ \#03_{size}$ **= 16 function points**

11.2.4. Problem count of RWP for FUR #04: Download Candidate Data

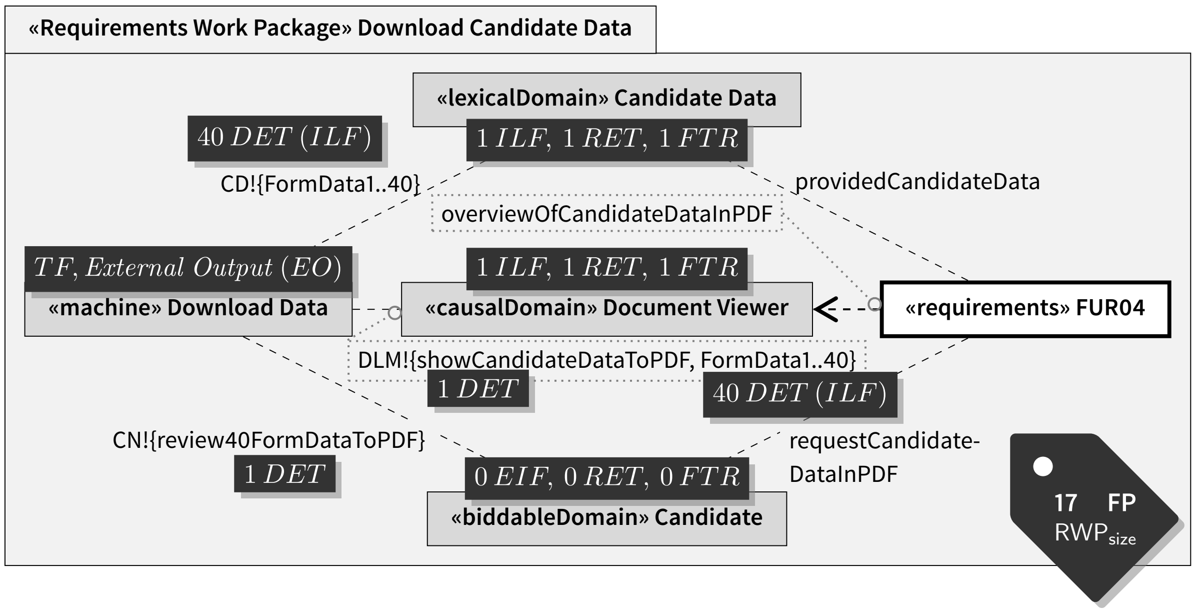

«Requirements Work Package» Download Candidate Data
«lexicalDomain» Candidate Data
40 DET (ILF)
1 ILF, 1 RET, 1 FTR
CD!{FormData1..40}
providedCandidateData
overviewOfCandidateDataInPDF
TF, External Output (EO)
1 ILF, 1 RET, 1 FTR
«machine» Download Data
«causalDomain» Document Viewer
«requirements» FUR04
DLM!{showCandidateDataToPDF, FormData1..40}
1 DET
40 DET (ILF)
CN!{review40FormDataToPDF}
requestCandidate-DataInPDF
1 DET
0 EIF, 0 RET, 0 FTR
«biddableDomain» Candidate
17 FP
RWP size

FIGURE 11.9 Counting Requirements Work Package FUR #04: Download Candidate Data

Comments on counting process activity	Results of activity
1. Classify FUR by Functional Size Measurement Patterns.	
Applied validation conditions: V.i - V.iii	commanded data-based control
2. Determine Data Functions. 2.a Identify problem domains as data functions.	
Applied validation conditions: V.iv	2 data functions: Document Viewer and Candidate Data
2.b Classify data functions into ILF or EIF.	
Applied validation conditions: V.vi, V.vii, V.viii, V.x	1 ILF: Document Viewer 1 ILF: Candidate Data
2.c Count DET for each data function.	
Count symbolic phenonema Applied validation conditions: V.xi	$DET_{document\ viewer} = 40$ $DET_{candidate\ data} = 40$
2.d Count RET for each data function.	
Applied validation conditions: V.xii	$RET_{document\ viewer} = 1$ $RET_{candidate\ data} = 1$
2.e Determine functional complexity for data functions.	
There are two ILF $ILF_{DET} = \sum_{i=1}^{n} DET_{ILF_i} = DET_{document\ viewer} + DET_{candidate\ data} = 40 + 40$ and $ILF_{RET} = \sum_{i=1}^{n} RET_{ILF_i} = RET_{document\ viewer} + RET_{candidate\ data} = 1 + 1$. No EIF.	~~$ILF_{DET} = 80$~~ $ILF_{RET} = 2$
But $ILF_{DET} - k$, and $k = 40$ equal phenomena $ILF_{Complexity}(ILF_{RET}, ILF_{DET}) = ILF_{Complexity}(2, 40)$ $EIF_{Complexity}(EIF_{RET}, EIF_{DET}) = EIF_{Complexity}(0, 0)$ Applied validation conditions: V.xiii, V.xvi, V.xvii, V.xix, V.xx	$ILF_{DET} = 40$ $ILF_{Complexity} = average$ $EIF_{Complexity} = \{n/a\}$
2.f Determine functional size for data functions.	
$ILF_{Size}(ILF_{Complexity}, ILF) = ILF_{Size}(average, ILF) = 10$ $EIF_{Size}(EIF_{Complexity}, EIF) = EIF_{Size}(\{n/a\}, EIF) = 0$ Applied validation conditions: V.xxii, V.xxiii	ILF_{Size} = 10 function points EIF_{Size} = 0 function points
3. Determine Transactional Function.	

Comments on counting process activity	Results of activity
3.a Identify machine domain as transactional function.	
	transactional function: download data
3.b Classify transactional function as either EI, EQ, or EO.	
query Applied validation conditions: V.xxv	$TF_{type} = EO$
3.c Count FTR for transactional function.	
TF_{FTR} = n ILF + m EIF = 2 + 0 = 2. Applied validation conditions: V.xxvi	$TF_{FTR} = 2$
3.d Count DET for transactional function.	
Only candidate and document viewer phenomena are counted Applied validation conditions: V.xxix, V.xxx, V.xxxi	$TF_{DET} = 42$
3.e Determine functional complexity for transactional function.	
$TF_{Complexity}(TF_{Type}, TF_{FTR}, TF_{DET}) = TF_{Complexity}(EO, 2, 42)$ Applied validation conditions: V.xxxii	$TF_{Complexity} = high$
3.f Determine functional size for transactional function.	
$TF_{Size}(TF_{Complexity}, TF_{Type}) = TF_{Size}(high, EO)$ Applied validation conditions: V.xxxiii	TF_{Size} = 7 function points
4. Report Functional Size for FUR.	
$MeasurableProblem_{size} = ILF_{Size} + EIF_{Size} + TF_{Size} = 10 + 0 + 7 = 17$ Applied validation conditions: V.xxxiv	$FUR\#04_{size}$ **= 17 function points**

11.2.5. Problem count of RWP for FUR #05: Upload Candidate Files

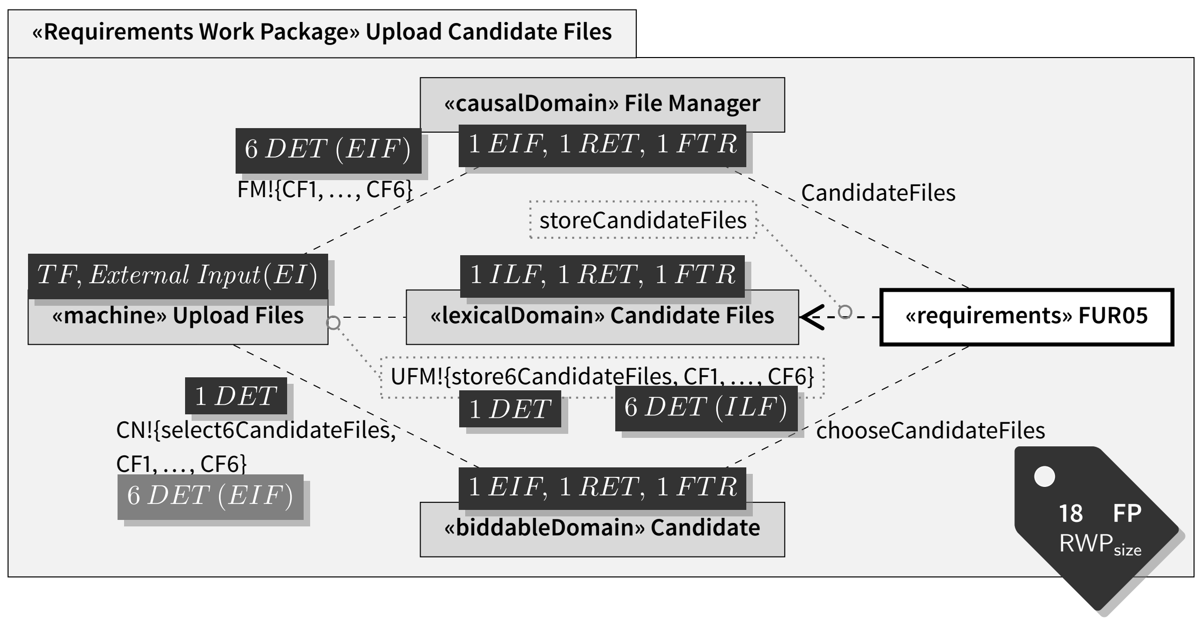


FIGURE 11.10 Counting Requirements Work Package FUR #05: Upload Candidate Files

Comments on counting process activity	Results of activity
1. Classify FUR by Functional Size Measurement Patterns.	
	commanded model building problem
2. Determine Data Functions. 2.a Identify problem domains as data functions.	
Applied validation conditions: V.iv	3 data functions: File Manager and Candidate Files and Candidate
2.b Classify data functions into ILF or EIF.	
Applied validation conditions: V.vi, V.vii, V.viii, V.ix, V.x	1 ILF: Candidate Files 2 EIF: File Manager and Candidate
2.c Count DET for each data function.	
	$DET_{file\ manager} = 6$ $DET_{candidate\ files} = 6$ $DET_{candidate} = 6$
2.d Count RET for each data function. 2.e Determine functional complexity for data functions.	$RET_{file\ manager} = 1$ $RET_{candidate\ files} = 1$ $RET_{candidate} = 1$
There is one ILF $ILF_{DET} = \sum_{i=1}^{n} DET_{ILF_i} = DET_{candidate\ files} = 6$ and $ILF_{RET} = \sum_{i=1}^{n} RET_{ILF_i} = RET_{candidate\ files} = 1$. There are two EIF $EIF_{DET} = \sum_{i=1}^{n} DET_{EIF_i} = DET_{file\ manager} + DET_{candidate} = 6 + 6 = 12$ and $EIF_{RET} = \sum_{i=1}^{n} RET_{EIF_i} = RET_{file\ manager} + RET_{candidate} = 1 + 1 = 2$. But $EIF_{DET} - k$, and $k = 6$ equal phenomena $ILF_{Complexity}(ILF_{RET}, ILF_{DET}) = ILF_{Complexity}(1, 6)$	$ILF_{DET} = 6$ $ILF_{RET} = 1$ ~~$EIF_{DET} = 12$~~ $EIF_{RET} = 2$ $EIF_{DET} = 6$ $ILF_{Complexity} = low$

Comments on counting process activity	Results of activity
$EIF_{Complexity}(EIF_{RET}, EIF_{DET}) = EIF_{Complexity}(2, 6)$ Applied validation conditions: V.xiii, V.xiv, V.xv, V.xvii, V.xviii, V.xx, V.xxi	$EIF_{Complexity} = low$
2.f Determine functional size for data functions.	
$ILF_{Size}(ILF_{Complexity}, ILF) = ILF_{Size}(low, ILF) = 7$ $EIF_{Size}(EIF_{Complexity}, EIF) = EIF_{Size}(low, EIF) = 5$ Applied validation conditions: V.xxiii, V.xxiv	$ILF_{Size} =$ 7 function points $EIF_{Size} =$ 5 function points
3. Determine Transactional Function. 3.a Identify machine domain as transactional function.	
	transactional function: upload files
3.b Classify transactional function as either EI, EQ, or EO.	
commanded model building	$TF_{type} = EI$
3.c Count FTR for transactional function.	
TF_{FTR} = n ILF + m EIF = 1 + 2 = 3.	$TF_{FTR} = 3$
3.d Count DET for transactional function.	
Only candidate and file manager phenomena are counted Applied validation conditions: V.xxvii, V.xxviii, V.xxix, V.xxx, V.xxxi	$TF_{DET} = 13$
3.e Determine functional complexity for transactional function.	
$TF_{Complexity}(TF_{Type}, TF_{FTR}, TF_{DET}) = TF_{Complexity}(EI, 3, 13)$	$TF_{Complexity} = high$
3.f Determine functional size for transactional function.	
$TF_{Size}(TF_{Complexity}, TF_{Type}) = TF_{Size}(high, EI)$	$TF_{Size} =$ 6 function points
4. Report Functional Size for FUR.	
$MeasurableProblem_{size} = ILF_{Size} + EIF_{Size} + TF_{Size} = 7 + 5 + 6$	$FUR\#05_{size} =$ **18 function points**

11.2.6. Problem count of RWP for FUR #06: Compile Candidate Résumé

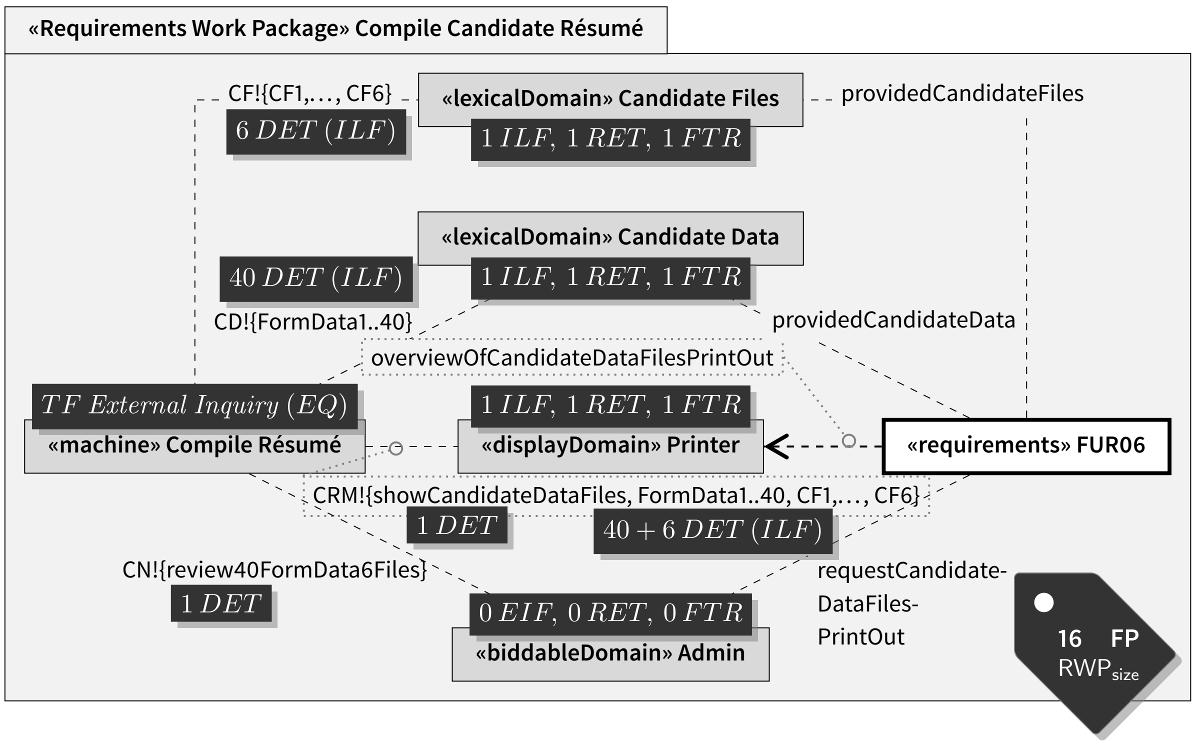

«Requirements Work Package» Compile Candidate Résumé
CF!{CF1,…, CF6}
$6\ DET\ (ILF)$
«lexicalDomain» Candidate Files
$1\ ILF,\ 1\ RET,\ 1\ FTR$
providedCandidateFiles
«lexicalDomain» Candidate Data
$40\ DET\ (ILF)$
$1\ ILF,\ 1\ RET,\ 1\ FTR$
CD!{FormData1..40}
providedCandidateData
overviewOfCandidateDataFilesPrintOut
$TF\ External\ Inquiry\ (EQ)$
«machine» Compile Résumé
$1\ ILF,\ 1\ RET,\ 1\ FTR$
«displayDomain» Printer
«requirements» FUR06
CRM!{showCandidateDataFiles, FormData1..40, CF1,…, CF6}
$1\ DET$
$40+6\ DET\ (ILF)$
CN!{review40FormData6Files}
$1\ DET$
$0\ EIF,\ 0\ RET,\ 0\ FTR$
«biddableDomain» Admin
requestCandidate-DataFiles-PrintOut
16 FP
RWP size

FIGURE 11.11 Counting Requirements Work Package FUR #06: Compile Candidate Résumé

Comments on counting process activity	Results of activity
1. Classify FUR by Functional Size Measurement Patterns.	
	query problem
2. Determine Data Functions. 2.a Identify problem domains as data functions.	
Applied validation conditions: V.iv V.iv	3 data functions: Candidate Files and Candidate Data and Printer
2.b Classify data functions into ILF or EIF.	
Applied validation conditions: V.vi, V.vii, V.viii, V.ix, V.x	3 ILF: Candidate Files and Candidate Data and Printer
2.c Count DET for each data function.	
	$DET_{candidate\ files} = 6$ $DET_{candidate\ data} = 40$ $DET_{printer} = 46$
2.d Count RET for each data function.	
	$RET_{candidate\ files} = 1$ $RET_{candidate\ data} = 1$ $RET_{printer} = 1$
2.e Determine functional complexity for data functions.	
There are three ILF $ILF_{DET} = \sum_{i=1}^{n} DET_{ILF_i} = DET_{candidate\ files} + DET_{candidate\ data} + DET_{printer} = 6 + 40 + 46$ and $ILF_{RET} = \sum_{i=1}^{n} RET_{ILF_i} = RET_{candidate\ files} + RET_{candidate\ data} + RET_{printer} = 3$. There are no EIF.	$ILF_{DET} = 92$ $ILF_{RET} = 3$

Comments on counting process activity	Results of activity
But $ILF_{DET} - k$, and $k = 46$ equal phenomena $ILF_{Complexity}(ILF_{RET}, ILF_{DET}) = ILF_{Complexity}(3, 46)$ $EIF_{Complexity}(EIF_{RET}, EIF_{DET}) = EIF_{Complexity}(0, 0)$ Applied validation conditions: V.xiii, V.xiv, V.xv, V.xvii, V.xviii, V.xx, V.xxi	$ILF_{DET} = 46$ $ILF_{Complexity} = average$ $EIF_{Complexity} = \{n/a\}$
2.f Determine functional size for data functions.	
$ILF_{Size}(ILF_{Complexity}, ILF) = ILF_{Size}(average, ILF) = 10$ $EIF_{Size}(EIF_{Complexity}, EIF) = EIF_{Size}(\{n/a\}, EIF) = 0$ Applied validation conditions: V.xxiii, V.xxiv	$ILF_{Size} =$ 10 function points $EIF_{Size} =$ 0 function points
3. Determine Transactional Function. 3.a Identify machine domain as transactional function.	
	transactional function: Compile Résumé
3.b Classify transactional function as either EI, EQ, or EO.	
query	$TF_{type} = EQ$
3.c Count FTR for transactional function.	
TF_{FTR} = n ILF + m EIF = 3 + 0 = 3.	$TF_{FTR} = 3$
3.d Count DET for transactional function.	
Only admin and printer are counted Applied validation conditions: V.xxvii, V.xxviii, V.xxix, V.xxx, V.xxxi	$TF_{DET} = 48$
3.e Determine functional complexity for transactional function.	
$TF_{Complexity}(TF_{Type}, TF_{FTR}, TF_{DET}) = TF_{Complexity}(EQ, 3, 48)$	$TF_{Complexity} = high$
3.f Determine functional size for transactional function.	
$TF_{Size}(TF_{Complexity}, TF_{Type}) = TF_{Size}(high, EQ)$	$TF_{Size} =$ 6 function points
4. Report Functional Size for FUR.	
$MeasurableProblem_{size} = ILF_{Size} + EIF_{Size} + TF_{Size} = 10 + 0 + 6 = 16$	$FUR\#06_{size} =$ **16 function points**

11.3. Use Case Decomposition

Details on the use case decomposition for the Student Recruitment Web Portal are given in section 8.6 on page 168, and figure 8.5 on page 168.

11.4. Requirements Specification

This section lists all task scenarios (as introduced and applicable for Problem templates in chapter 7.4) for the requirements **FUR #01** to **FUR #06** of the Student Recruitment Web Portal.

11.4.1. Task scenarios of FUR #01: Grant Access Authorization

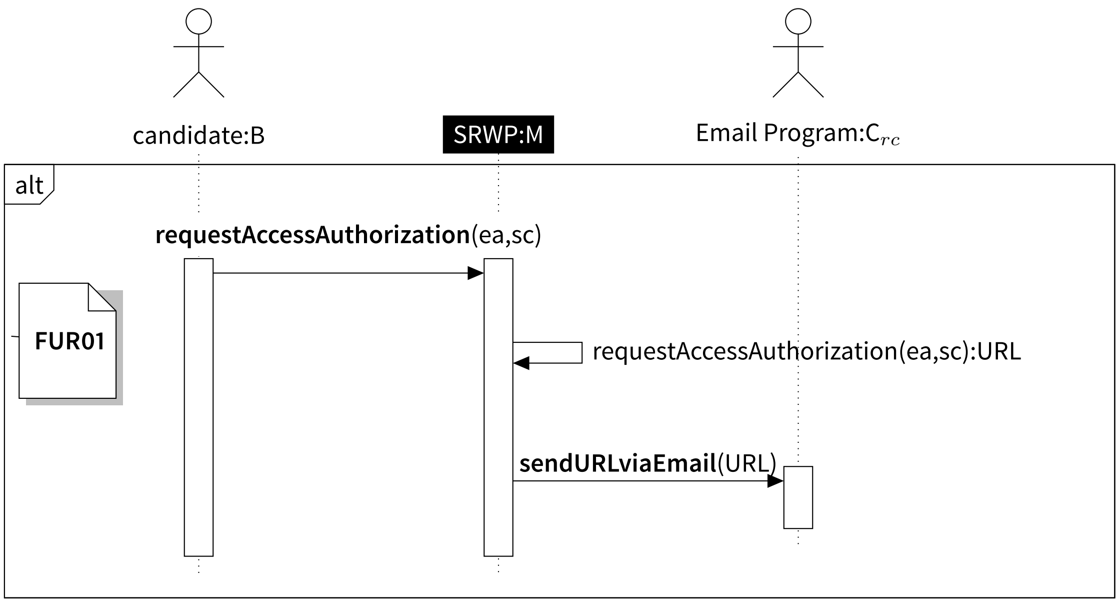

candidate:B
SRWP:M
Email Program:C_{rc}
alt
FUR01
requestAccessAuthorization(ea,sc)
requestAccessAuthorization(ea,sc):URL
sendURLviaEmail(URL)

FIGURE 11.12 Specification of RWP for FUR #01: Grant Access Authorization

11.4.2. Task scenarios of FUR #02: Record Candidate Data

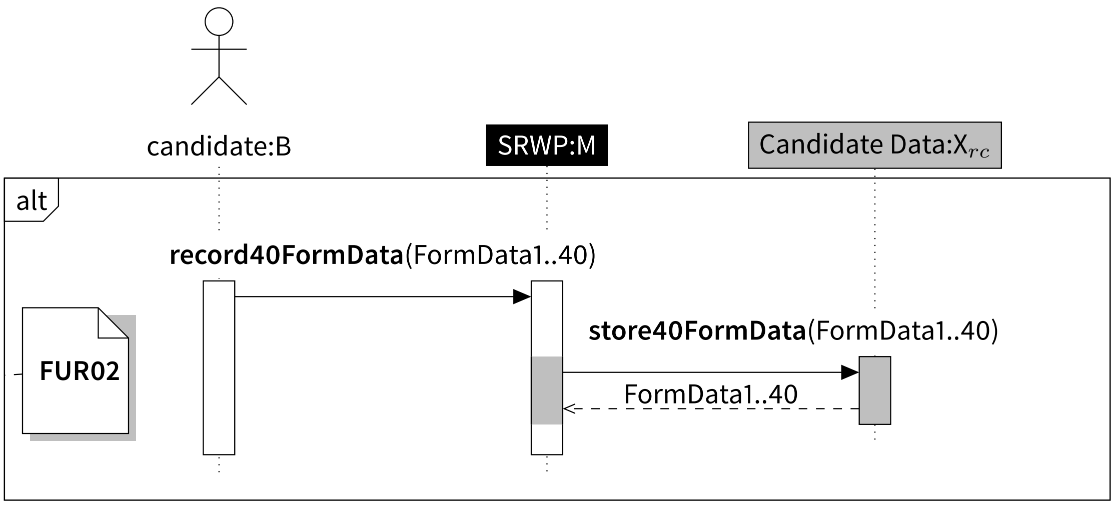

candidate:B
SRWP:M
Candidate Data:X_{rc}
alt
FUR02
record40FormData(FormData1..40)
store40FormData(FormData1..40)
FormData1..40

FIGURE 11.13 Specification of RWP for FUR #02: Record Candidate Data

11.4.3. Task scenarios of FUR #03: Review Candidate Data

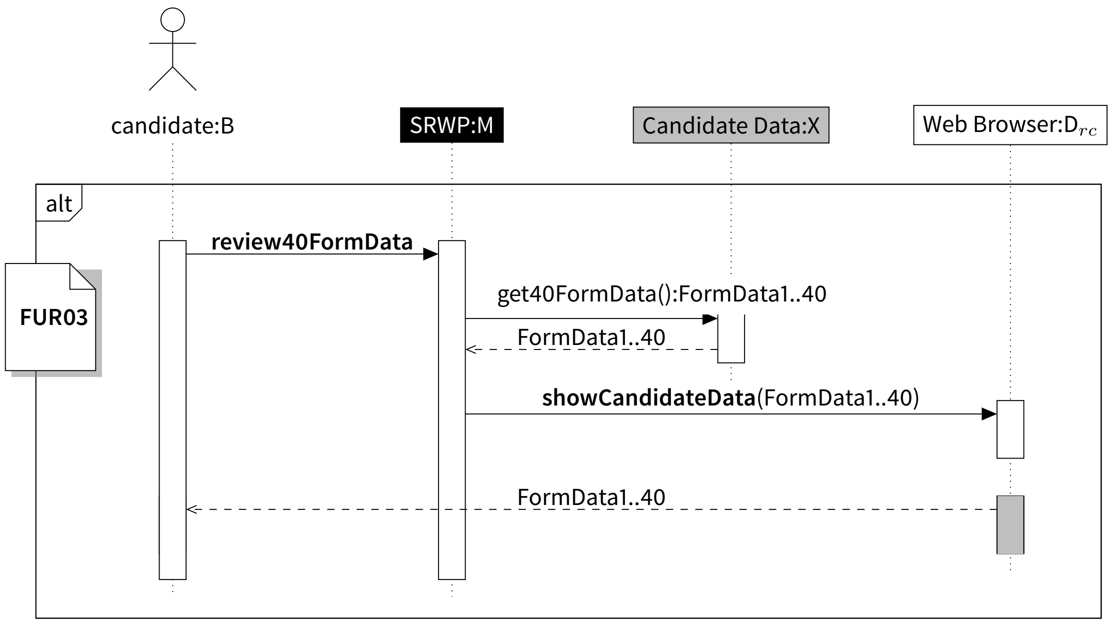


FIGURE 11.14 Specification of RWP for FUR #03: Review Candidate Data

11.4.4. Task scenarios of FUR #04: Download Candidate Data

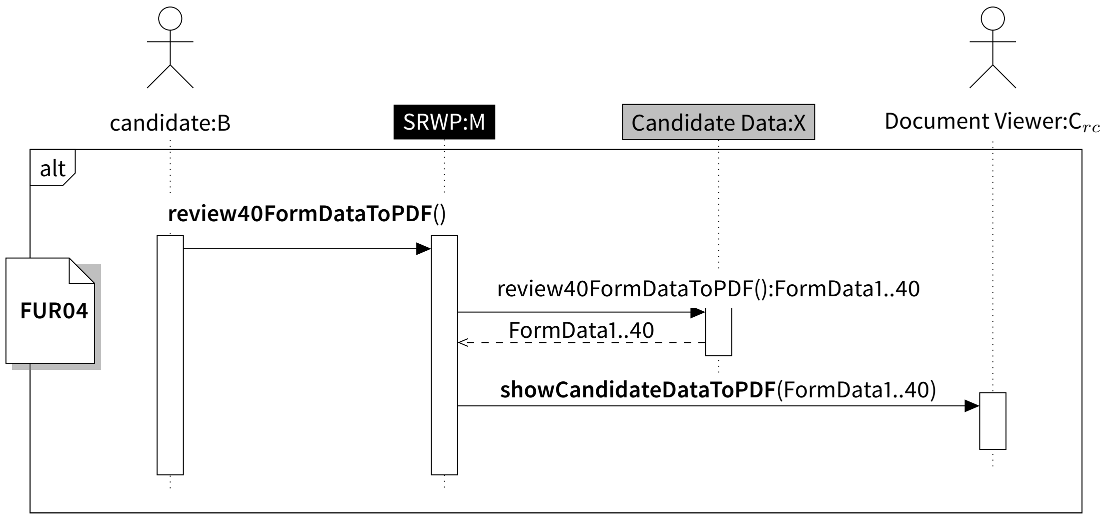


FIGURE 11.15 Specification of RWP for FUR #04: Download Candidate Data

11.4.5. Task scenarios of FUR #05: Upload Candidate Files

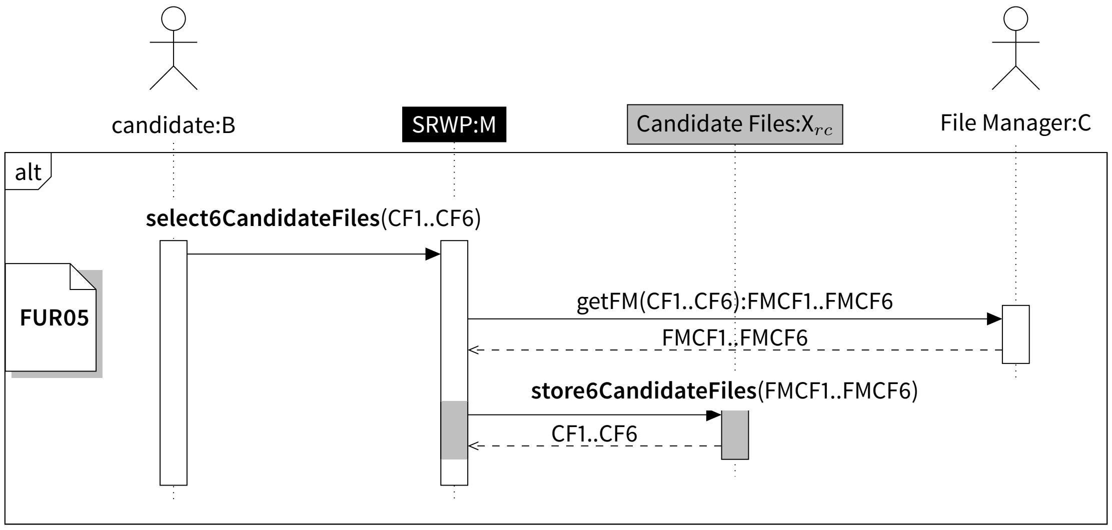

candidate:B
SRWP:M
Candidate Files:X_{rc}
File Manager:C
alt
FUR05
select6CandidateFiles(CF1..CF6)
getFM(CF1..CF6):FMCF1..FMCF6
FMCF1..FMCF6
store6CandidateFiles(FMCF1..FMCF6)
CF1..CF6

FIGURE 11.16 Specification of RWP for FUR #05: Upload Candidate Files

11.4.6. Task scenarios of FUR #06: Compile Candidate Résumé

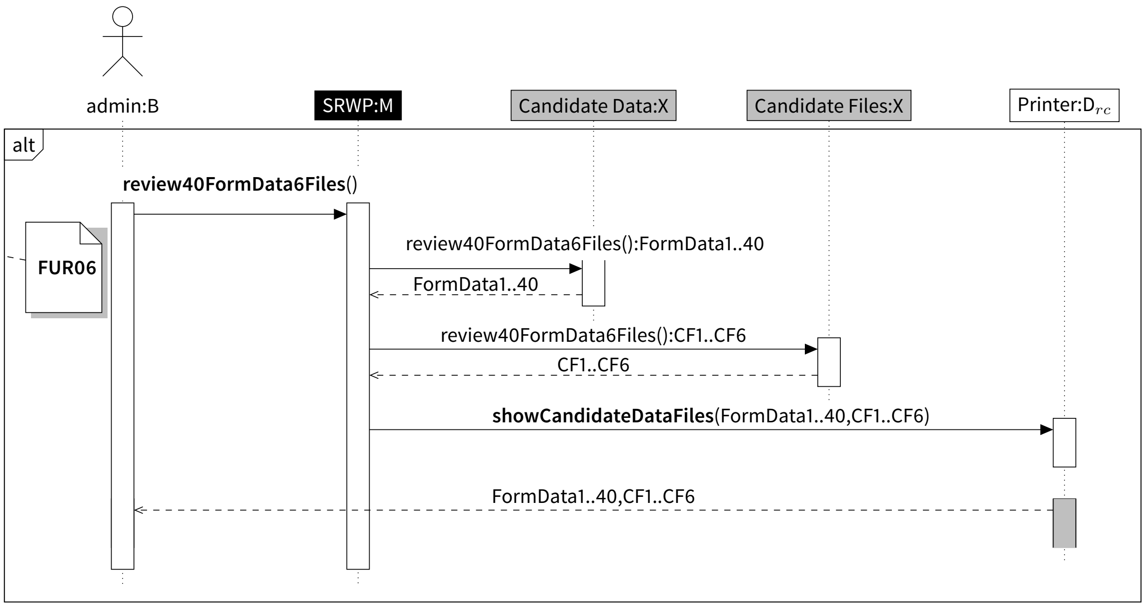

admin:B
SRWP:M
Candidate Data:X
Candidate Files:X
Printer:D_{rc}
alt
FUR06
review40FormData6Files()
review40FormData6Files():FormData1..40
FormData1..40
review40FormData6Files():CF1..CF6
CF1..CF6
showCandidateDataFiles(FormData1..40,CF1..CF6)
FormData1..40,CF1..CF6

FIGURE 11.17 Specification of RWP for FUR #06: Compile Candidate Résumé

11.5. Requirements Dependencies

In this section, the dependencies of requirements that belong to different measurable problems are further elaborated with the aim to support requirements prioritization, which not only contributes to determining the capacity of a Project Backlog and product owner value, but also impacts the choice of a product backlog item, which is planned for the next Project Time-Box.

With regard to the product backlog of the student recruitment web portal, table 11.3 on page 226 shows how the life-cycle expressions for this application as given on page 167 control the priority of each item.

Considering these requirements dependencies and their functional size leads in this case to the suggestion to add $FUR\ \#01\ Grant\ Access\ Authorization$ at next to the project backlog. The priority of this product backlog item is ❶, since it involves software functionality that occurs at first in the life-cycle for this application. In contrast to this, the product backlog items FUR #03 and FUR #04 share the lowest priority of ❺, since they represent optional software functionality regarding the life-cycle expressions. It is easy to agree to plan mandatory features before nice to have ones.

11.5.1. Life-Cycle Expressions

The life-cycle expression for the student recruitment web portal relates the requirements work packages FUR01 to FUR06 as follows:

$$\begin{aligned} LC_{admin} = \;& FUR06.review40FormData6Files \\ LC_{candidate} = \;& FUR01.requestAccessAuthorization; \\ & FUR02.record40FormData; \\ & [FUR03.review40FormData \,||\, FUR04.review40FormDataToPDF]; \\ & FUR05.select6CandidateFiles \end{aligned}$$

$$LC_{student\ recruitment\ web\ portal} = (||_{i=1}^{n} LC_{admin_i}) \,||\, (||_{j=1}^{m} LC_{candidate_j})$$

The optional expression on $LC_{candidate}$ is of special use for requirements prioritization. It indicates that the requirements work packages FUR03 and FUR04 are of less relevance to a successful completion of a candidate's application than the other requirements work packages. This gives reasons for considering the others first in project planning, i.e. to include FUR03 and FUR04 at last to the Project Backlog. LC_{admin} and $LC_{candidate}$ are synchronized with respect to the degree of completion for a candidate's application, which models the states of the student recruitment web portal in figure 8.4.

11.5.2. State Transition Diagram

Details on requirements synchronization for the Student Recruitment Web Portal are given in section 8.5 and page 165, and by figure 8.4 on page 167.

Part VI.

Epilogue

Part VI Epilogue compiles the findings and implications of taking advantage from pre-defined units for planning the scope and speed in software projects as investigated by this dissertation. Chapter 12 Conclusion summarizes the answers to the research questions as detailed in section 1.2. Chapter 13 Future Prospect outlines remaining issues and newly found directions for paving the way for worthwhile research, one which contributes to a sustainably managed software engineering discipline in software development projects.

12. Conclusion

Problem-Based Project Planning is about pattern-enhanced, size-driven practices to meet *the Need for Speed* as is attributed to postmodern software engineering projects. It aims at empowering project teams to control project success effectively and sustainably. For this purpose, recognizable units of shareable problem-solving knowledge are introduced, which serve the team to frame their common understanding on the project tradeoffs to be made and thus to speed up project progress.

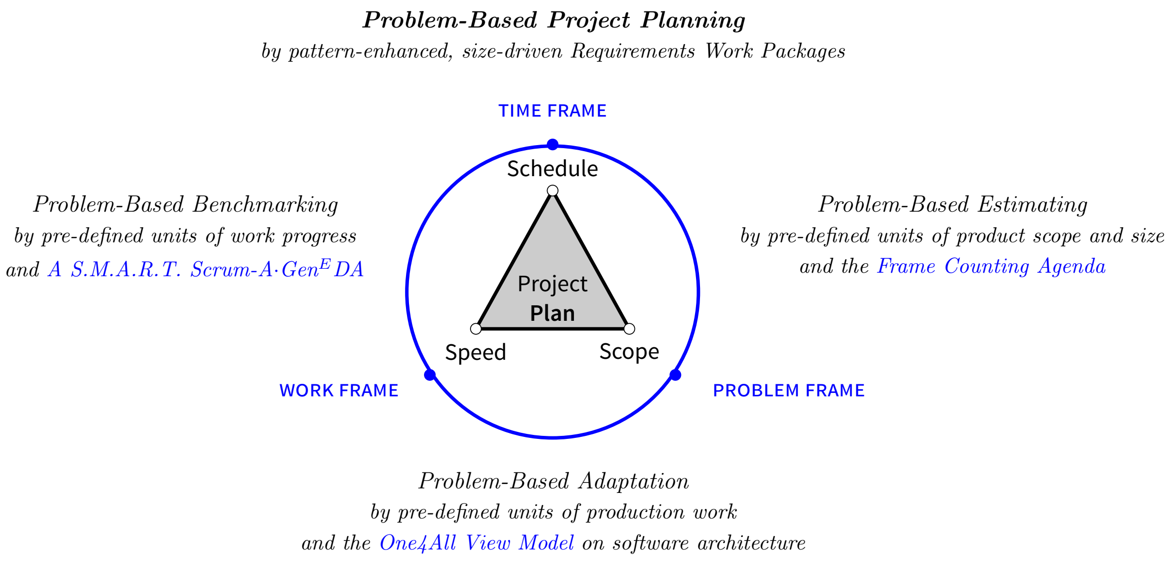

Problem-Based Project Planning
by pattern-enhanced, size-driven Requirements Work Packages
TIME FRAME
Schedule
Problem-Based Benchmarking
by pre-defined units of work progress
and A S.M.A.R.T. Scrum-A·Gen^{E}DA
Problem-Based Estimating
by pre-defined units of product scope and size
and the Frame Counting Agenda
Project
Plan
Speed
Scope
WORK FRAME
PROBLEM FRAME
Problem-Based Adaptation
by pre-defined units of production work
and the One4All View Model on software architecture

FIGURE 12.1 A Postmodern Software Engineering Approach to Project Planning

12.1. Problem-Based Enablement of Agile Software Engineering Projects

Postmodern software engineering projects rely on empirism, which “asserts that knowledge comes from experience *and* making decisions based on what is known” [193, page 5]. That is, the project team depends on sharing their “empirical observation [...of] what works” [86].

Improving the common understanding of what is known by a project team, makes it agile, as it becomes “better able to adapt” [86] to the fuzziness involved with (requirements) change. Agile project teams have a high responsiveness to change, as they are capable to draw value from their lessons learned.

For this, agile project practices encourage transparency, inspection, and adaptation, which are the three pillars of empirical process control [193, page 5]. Problem-Based Project Planning takes account of these, thereby answering the research questions of this dissertation.

12.1.1. Transparency

Transparency [193, page 5] means to make the common understanding of the significant aspects in the project visible, by defining a standard or language that is shared by all members of the team.

This dissertation considers functional user requirements as those significant aspects in the project, which are of most importance to the team.

As summarized in figure 12.1 by the contribution **C 01** Problem-Based Functional Size Measurement Patterns it introduces a means for structuring desired software functionality into Requirements Work Packages. These pre-defined units of *product scope and size* allow for framing the team members' understanding of what should be achieved in the project to a recognizable level of detail and type of functionality. They equip the team with a standardized view model on the problems, they are going to solve in the project, and focus respective knowledge sharing on what is affected by a change. As a result of that, the modeling of software product requirements which builds on problem-based functional size measurement patterns presents an answer to the research question **RQ 1.a** How to establish pre-defined units of scope?

These units not only enable the team to fix the scope of requirements, but they are also applicable for determining consistent estimates on how to satisfy these and when to expect the completion of respective work on these. Both of these estimates are essential inputs to the project planning. Problem-Based Project Estimating is a prerequisite for measuring project product as well as production performance. Both is made possible by providing the team with contribution **C 02** Problem-Based Functional Size Measurement Method named Frame Counting Agenda, which is the answer to research question **RQ 1.b** How to estimate scope size?. It makes transparent what counts in and what does not for determining the size of a problem and the speed of delivering its solution.

12.1.2. Adaptation

Adaptation [193, page 5] means to react to visible deviations of significant aspects in the project instantly, in case these would cause project results to become unacceptable.

In this dissertation, deviations of significant aspects in the project are considered as change in requirements, one which is not visible to the team in the project plan. This deviation manifests itself in the team's inability to act as planned or in its failure to deliver a working solution within the project time available.

In order to prevent the team for suffering from unwanted changes that if not properly adjusted, will creep the scope of a project plan, Problem-Based Project Adaptation as one pillar of problem-based project planning according to figure 12.1, exploits requirement dependencies in two ways:

First, it accounts for the structural dependencies of requirements and software architecture, thereby providing an answer for **RQ 2.a** How to establish pre-defined units of work?
As agile project principles demand for *Embracing change*, the team is supported by contribution **C 03** Transition Templates for bridging the gap between problem analysis and solution design. This newly developed kind of patterns qualify Requirements Work Packages as boundary objects, which are meaningful to the product and production perspective in a project. They guide the team in determining a plan of *production work* that fits the product scope as defined in each requirements unit, i.e. one that lets the team produce desired project results. In case of emergent requirements, which impede progress as planned and thus hinder expected project performance, the team is made ready for adapting its planning by deciding instantly on alternative plans on how to proceed with the project.

Second, it deals with the functional dependencies of requirements and software architecture for answering the question **RQ 2.b** How to plan worthwhile work volume?
Agile project procedure comes with a fixed time frame available for producing desired results. As each project team has a limited capacity of production work they can do in time, which implies that they cannot do everything at once, agile teams strive towards the principle of *Maximize the work not done* flexing the what is done in the project. The Processes View as part of contribution **C 04** "One4All" View Model on Software Architecture supports the team in having control over their projected work volume. It facilitates the prioritization of Requirements Work Packages according to their value for the product's users. The knowledge about what should be done first is inherent to the user's business processes and related workflow models. This dissertation uses the software lifecycle as a source of information for deciding on a worthwhile volume of work that takes into account the priorities of the requirements.

12.1.3. Inspection

Inspection [193, page 5] means to frequently compare project progress against its plan for detecting those significant aspects in the project, which run outside acceptable limits.

As "a process is only as good as is the *rigour* of its application" [184, page 366], this dissertation integrates problem-based project planning and an agile project process framework to A S.M.A.R.T. Scrum-A·Gen^{E}DA. It defines the time intervals within a project in which inspections should take place.

The basis for comparison is given by contribution **C 05** Problem-Based Project Baseline, which is represented by a Project Backlog that builds on Requirements Work Packages as work items. This Project Backlog makes the project plan. It is a time-bound view on selected items from the Product Backlog, which answers the research question **RQ 3.a** How to baseline project plans?

Project progress against this plan can be inspected as soon as work on the Project Backlog has started. Those work items "done" within the project time available are used as **C 06** Problem-Based Speed Benchmark. As each is assigned with a product size measurement given in function points, these become valuable units of progress applicable for Problem-Based Project Benchmarking.

The answer to the research question **RQ 3.b** How to benchmark the progress of projects? lies in the backlogs as established by Problem-Based Project Planning. These preserve the speed measurements and respective lessons learned of different projects and teams, and make their project achievements compare- and reusable. This sustainable best practices knowledge empowers teams to accelerate their decision making for the benefit of project progress.

13. Future Prospect

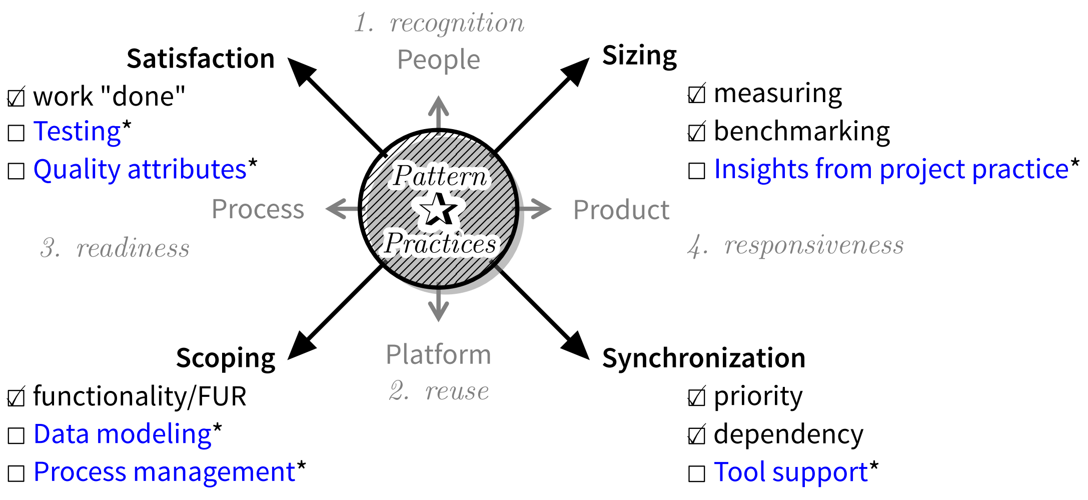
13.1. How *fast* can the software project team become?

As refined by the research questions of this dissertation, it depends on the approach how the speed of software project teams can be measured comparably, and how their speed can be controlled reproducibly.

Comprehensible performance data that reveals speed differences as well as the underlying causes of these is needed for judging how fast the software project team actually is and still can become. Knowing their speed and the practices involved with it are prerequisites for the continuous improvement processes undertaken in software projects and by their teams.

As “projects prosper to the extent that people learn to work together effectively.” [75, page 56], it matters to which extent the team can take advantage from their lessons learned for executing a project under specific conditions. Only then, “enhanced performance by learning” [113, page 1] becomes possible.

Making this know-how available to the project team is based on pattern practices in this work. It is made retrievable via equally scoped and sizeable requirements work packages, which serve as pre-defined, multi-purpose work items to the project team.



☆ = **Sustainability** is a key to speed
☑ = subject to this dissertation
* = future research areas

FIGURE 13.1 Sustainable decision-making and speed improvement practices enable accelerated performance

Figure 13.1 illustrates how this work evolves McConnell's four dimensions of software project speed [151] in each dimension of people, product, platform and process by the use of pattern practices for the sake of systematic speed consideration and its improvement in software projects.

Patterns at the people dimension support the *recognition* of recurring problems, which leverages collaboration in the team. At the platform dimension, patterns guide the selection and *reuse* of proven solutions, which helps to speed up the realization of desired software functionality and "minimizes wasted effort" [206, chapter 6].

In both of these dimensions, patterns *stabilize* the project team's common understanding of what the user wants, and allow for an increased anticipation of the work required to respond those needs properly. That way empowered teams can start sooner, as the project plan becomes instantly available to them.

The use of patterns at the process dimension increases the *readiness* of the project team to overcome obstacles in regard to their responsibilities. Patterns are boundary objects, which not only resolve replication of errors and work, but also reduce lead times caused by handovers, which usually hinder projects from progress(ing fast(er)). They pave the way for technical excellence, the best guarantee against technical debt, and its associated risk of performance degradation. At the product dimension, patterns support the *responsiveness* of a team to react proactively and not reactively to (changed) user needs, and for focusing and flexing "their attention to doing the work that is of highest value to the customer" [206, chapter 1] for deciding on what has to be done first.

In both of these dimensions, patterns help to *separate* responsibilities and results despite increasing complexity and variability, which comes with change. They establish a separation of concerns, which on the one hand maintains the relation among functional user requirements, and on the other hand makes an independent consideration of associated work items possible. Pattern practices contribute to a faster adaptability of the work plan in regard to its prioritization and volumne, one that really lets teams 'embrace change'.

Functional user requirements stabilized and separated by patterns (practices) form the basis for creating adaptable software products and project plans. These enable comprehensible decision-making, which is essential for delivering value to the customer at a sustainable pace, and for having reason(able) to 'trust (in) the team'.

13.2. Future Directions – Towards Sustainable Software Engineering Practice

This section gives some details on the future research areas as noted in Figure 13.1. Because some time has passed between writing this dissertation and delivering it, experiences made by the author with the herein proposed approaches is reported in each example paragraph in the following. The effects on the results of this work and their improvement due to their application in project practice are explained in more detail in Section F For Further Discussion. The examples come from the field of software development projects for form dialog-based web applications to support administrative processes.

13.2.1. Insights from project practice

As Figure 13.1 shows, this dissertation deals with the measurement and benchmarking of functional user requirements, with the focus on their functional size. The intention is to replace gut feel estimates in software development planning by a counting method in order to make comparable software product and production process data available. This data helps projects and their teams in determining (the reason for) their speed, which is needed for building a work plan, they can have confidence in. For identifying comparable data and the related best practices, the approach followed in this dissertation depends on the application of analogies [96, 199, 210] for the recognition of commonalities in software functionality, which is provisionable by patterns.

The interesting question is here: *How applicable are these patterns in real software projects?* Which are the challenges to adapt them to different

- project types,
  such as green field ("from scratch"[1]) or brown field ("evolutionary"[2]) development, or to
- project organizations and processes,
  for instance regarding standards such as ITIL [20], PMBOK [124], PRINCE2 (Agile) [19, 21], or V-Modell [77]?

There is good and bad news: The use of problem-based functional size measurement patterns for measuring functional size is less interesting than their use for scoping software products in practice. It turned out that the reproducible separation of functional user requirements into independent but still interrelatable requirements work packages, which is also made possible by problem-based functional size measurement patterns is the more significant and needed [143] contribution.

EXAMPLE 13.1 Agile Modernization

Within several software development projects, the concept and use of problem-based functional size measurement patterns turned out to be very supportive for the requirements decomposition and respective identification of reusable software artefacts.
These patterns have been developed further into six problem-based user story templates for recognizing recurring classes of functional user requirements by adopting the concept of *requirements templates* and *process words* from [207]. Each template makes use of a specific keyword for identifying its transactional function (TF), see Table F.1 Problem-based user story templates applying TF-keywords on page 291. Especially for modernization projects of highly customized legacy software, where the knowledge about a product's functionality and its respective documentation is not unambiguously available anymore, user story templates contribute to the systematic execution of reengineering activities and care for requirements completeness.
After redocumenting several legacy products (on the fly in agile settings, as well as retrospectively in waterfall-like projects) by the help of user story templates, it turned out that not the measured size of each, but already the total number of resulting work packages contibutes to the projects' comparability in a convincingly reliable way. That way, a correlation between amount of requirements work packages and the time needed for delivering these could been observed across different projects, and successfully applied for their planning. Furthermore, the recognition of recurring problems and their solution alternatives has allowed to compare different implementations of software artefacts and to decide whether they should be retained or how to replace them.
This was made possible by the user templates' underlying pattern practices, which establish equally scoped units of (comparable kinds of) software functionality.

[1]green field = newly set up, homegene environments with expectable low integration effort, and the assumption of manageable, since known dependencies

[2]brown field = grown, heterogeneous environments with expectable high integration effort, and the risk of unknown dependencies

13.2.2. Tool support

As Figure 13.1 shows, this dissertation deals with the dependency and priority of functional user requirements for enhancing responsiveness of software project output to user needs. The intention is to keep demand synchronized with development in order to proactively control project progress towards desired directions under changeable conditions.

One cause of divergency between demand and development in the presence of change is the “Barely Sufficient” Documentation [206, chapter 4] as is commonly observable in software development projects. Documentation activities are too often perceived as time consuming necessary evil, whose contribution to the output of the project namely the software product is undervalued. Tool support which takes advantages from pattern practices can ease the creation and maintainability of a living software (requirements) documentation, one which makes sure the constant encouragement of the people involved in a project by improving their mutual understanding and concerning their individual work contribution.

A cascading transition between documents from different domains or disciplines based on patterns is an important first step in order to create synergies, e.g. to systematically combine the requirements analysis with the architecture design and the subsequent software development in a forward and backward navigable manner.

The next step towards a living documentation is to use patterns in such a way that they serve as anchors to which appropriate documentation from different disciplines can be attached immediately. Not just the patterns, but the resulting document instances as well should be understandable and purposeful ’cases’ in different contexts. This approach is an implementation of “projectional editing” as described by Fowler [87]. For example, a screenshot of a prototype is sometimes a better product specification than a wall of text in natural language, because it is understandable and of purpose to (guide) its users and developers (in fulfilling their responsibilities in the project). The anchor, ie. what matters to both is still framed by the underlying pattern. This allows the cascade of transitions to be skipped, and to reduce the risk of losing information and adding content, like in a ’chinese whisper’ game. It would accelerate the implementation of changes and the corresponding adaptation of the documentation.

So, *how integrate the pre-defined units of software product size and scope as proposed in this dissertation with exisiting tools for determining work progress and for keeping up a living documention of the software product(ion work)?*

The significant added value of requirements work packages is their pre-defined, equally sliced functional scope as a result of the underlying problem-based functional size measurement patterns, which makes an independent consideration of these units of “basic activities” for different purposes in software projects possible. Tooling support for the

- project process,

  for example ScopeMaster [195], which assists determining the functional size in requirements analysis, or a work assignment system such as Atlassian Jira [18], which provides tickets applicable as story cards for organizing the work within software development Sprints, and options for limiting work-in-progress by Boards in kanban-style, would benefit from the pre-defined units of software functionality as proposed in here

  - for streamlining their definition of functional scope and for executing respective segmentation in the requirements analysis and story specification, and
  - for normalizing their key figures in accordance with the number of requirements work packages and their respective functional size to obtain a comparable measurement for and definition of (work) done.

Tooling support for the

- product documentation,

  can take advantage of the pre-defined logical boundary that comes with each problem-based functional size measurement pattern by using it for the (automatable) linkage of self-contained software functionality, which is either specified as requirements artefacts or given as software code within a *Biz*Dev*Sec*[3]Ops-like toolchain[4]. In this context, research can be

  - started by investigating the ((requirements) work) packages' integrability with scripting approaches such as the Gherkin language [72] for creating an executable requirements specification[5], or the application of visual modeling[6] tools of user dialogs as Balsamiq [27] for creating click-through prototypes. These help the project team to get a first impression of the feasibility of requirements based on an early project phase's product documentation. Paragraph **EXAMPLE 13.2** reports on an attempt to make this work (out).
  - continued in the direction of technological frameworks such as JAVA microservices [183], as these share the concept pursued in this dissertation: to set up recognizable service units with a single processing responsibility that is bound to a defined data set. The packing of software product (documentation) artefacts into self-contained units of software functionality can be continued along the toolchain by containerization, as is possible by Docker [81]. This increases flexibility in later phases of software development projects, as the orchestration of software product (service) artefacts is more adaptable to changing user needs. The logical boundary of artefacts at the level of software code and requirements specification is the same. That way, the change of one is immediately understandable to the other.

EXAMPLE 13.2 Backlogs and Wikis

Using the same pattern-based framing for the functional scope of early and late software (documentation) artefacts and respective tool support, helps the project team to gather a common understanding of the problem to be solved and improves their collaboration on its solution.
Since the project teams embody different perspectives and different disciplines, and its members have to fulfill different reponisibilities, it is important to focus on shareable "basic units of activity", because considerations about units where "There is only one WHEN event that triggers the scenario" [163, page 60] is not distracted or confused by problems outside and within their pre-defined logical boundaries [84]. This facilitates mutual agreement in the team on how to proceed.
For example, a structuring of Requirements Work Package as multi-purpose work item as shown in Table F.3 serves as a kind of agenda to the project team, which is comprehensible to users and developers, and guides their discussion of demand towards development during their project planning.
Requirements work packages, which are presented as story tickets by a tool such as Atlassian Jira [18], for example, can be used within a backlog to set up a software product baseline, and likewise integrated into a wiki such as Atlassian Confluence [18] for multi-documentation purposes, for example to make contributions to specification and acceptance documents, technical manuals and user training, etc. In this way, story tickets become multi-purpose work items that are not only used for planning, but also serve the decision-making in subsequent project process phases or Sprint events. These work items form an anchor that makes changes transparent and inevitably keeps documentation updated by those who depend on and work on them.

[3]"Quality by Design" issues are discussed in Section 13.2.5
[4]Without the intention of advertising or opportunity to use: Atlassian provides a plugin named test management for Jira (TM4J), which integrates Gherkin for establishing a DevOps-chain team collaboration.
[5]The user story templates in Table F.1 account for gherkin
[6]Visual programming serves also the realization process, in terms of knowledge building and implementation practice.

13.2.3. Process management

As Figure 13.1 shows, this dissertation primary deals with the behavioral aspects of functional user requirements. The intent is to frame user requirements by means of patterns into independent, measurable units of software functionality, which are comparable in their functional scope, and because of that assist the software product planning and its production process.

Based on these "basic units of activities", their flow of usage within a software product is modeled by life-cycle expressions, or alternatively by state-transition diagrams in order to systematically link them with one another into meaningful usage scenarios. This modeling takes into account the functional dependency of the user requirements. The priority of units or a particular flow of use depending on their value to the user or the developer of the software product.

Obviously, the further development of work flow modeling by notational means, which are closer to business and respective value considerations, such as BPMN [88, 165][7] and BABOK [163], is a feasable future research direction.

It seems more promising, however, to address the question of *how do requirements work packages affect the prioritization of work flow within (project business) processes of the product delivery pipeline, for example in the context of a problem-based*

- *Risk Management, or*
- *Resource Management?*

Section 13.2.6 Testing gives an outlook on problem-based risk management for the sake of user acceptance testing. The case of resource management in problem-based project planning is opened in the following paragraph **EXAMPLE 13.3**. In addition, the current approach to workflow modeling in problem-based project planning is presented, which has been further developed through use in different software development projects.

EXAMPLE 13.3 Boards and Capaci/bilities

The Tables F.8 Story Mapping for the Vacation Rentals Web Application and F.7 Story Mapping for the Student Recruitment Web Portal show an alternative workflow model compared to state-transition diagrams and life-cycle expressions for the Case Studies discussed in this dissertation.
These tables are not a 1:1-implementation of a Story Map [11], but they retain their concept to assist "in creating understanding of product functionality, the flow of usage, and to assist with prioritizing product delivery" [163, page 100].
A table row describes a "scenario [...] in terms of how stakeholders interact with the solution" [163, page 93], which is made visible by relating associated user (roles) in each (scenario) row, in addition to the flow of individual user stories. For this reason, the term *Storyboard* also fits this tabular representation of dependent software product functionalities.
This representation has several advantages over a stacked product backlog. For example, repeated appearance of a user story indicates its importance for the user and the respective priority for its development. In addition, a Storyboard provides the "big picture" of software product that is not visible in a Sprintboard. It only contains a time-boxed excerpt of user stories from the product backlog. It represents the user story map or *product roadmap*, too.

[7]The BPMN task types (receive, service, send) relate to IFPUG FSM elementary process types (input, inquiry, output) and the basic types of functionality in Problem-Based FSM (TOFF-i., TOFF-ii., TOFF-iii.)

EXAMPLE 13.4 Boards and Capaci/bilities, continued

The pre-defined units for Problem-Based Project Planning serves to size the scope of the product, i.e. its functional user requirements. This size measure (in function points) determines the project baseline and answers the question of how much work needs to be done?
In addition, this size measure is of use in the time dimension of project planning, as it can quantify how much work is actually being done.
The question that has not been addressed so far relates to the resources dimension of project planning: Do we have the capacity, i.e. the resources available to get the work done? When can the work be done?
The resources available can either relate to the capabilities, i.e. the skills of the project team, which are enhanced by pattern-enabled best practices in this dissertation, or the notion of resource can refer to the capacity, i.e. the timely availability of individual project team members to complete the work.
In this dimension of project planning, the functional dependency of individual requirements work packages from one another , which is addressed in this dissertation by workflow considerations and prioritization in order to fit the project time-box, is not important, but the dependence of individual requirements work packages on the timely availability of the project organization's functional units (represented by each project team member) is of relevance and worthwhile future research. The problem to be solved is to plan the availability of resources in the project time-box, so that the work can be done.
Planning the scope and speed of software production is one step. Planning the availability of skills during software production, i.e. the resource or team member that ultimately does the work, is the next.

13.2.4. Data modeling

As Figure 13.1 shows, this dissertation mainly deals with the functionality of user requirements to frame recognizable “basic units of activities” for establishing a requirements model that assists the software product planning and its production process.

The intention to address the data involved with functional user requirements belongs to the area of future research. Since problem-based functional size measurement patterns showed to be useful for separating functional user requirements into units of desired software functionality, the question at hand is: how does the associated logical boundary contribute not only to its functional scope, but also to that of data, which are also bound to these units?

How is the (structuring of) data that is linked to a requirements work package taken into accounted through different

- *notations and transformations*
  (UML [168], OCL [167], ER diagrams, etc.)
- *application domains and purposes*
  (technical documentation, data processing rights, test data generation)

for creating a data model?

EXAMPLE 13.5 Roles and Responsibilities

Data modeling is not negligible in problem-based functional size measurement, but it is not bound to a notation. As elements of so-called ’data functions’, data information, which is processed by a recognizable unit of software functionality, counts in to the overall function points that determine a requirements work packages’ functional size.
Table F.2 gives an idea of how the expressiveness of keywords, as used in user story templates, can be further developed through OCL expressions, to map data that is linked to requirements work packages (according to a target structure such as in Figure F.1) to a UML class diagram. Figure F.2 Exemplary Data Model for the Vacation Rentals shows, how this modeling approach is applied to the Vacation Rentals Case Study. This data model, specified as a UML class diagram, can be used for documentation and auditing purposes, for instance in the context of data protection and privacy analysis as outlined in Section 13.2.5 Quality attributes, and it offers options for integration into (automated) testing tools as presented in Section 13.2.6 Testing.
In addition, Table F.2 List of TF-keywords for problem-based user story templates assignes each TF-keyword read and write responsibilities for its data function.
For instance, a requirements work package which is linked to data for creating a user account, i.e. $TF{=}create$ and $DF/element{=}user\ account$, is responsible for including a new element into the collection of existing user accounts by executing a respective write operation to a data base.
Reading information related to this user account is not in focus of this requirements work package. That is why $TFK1.$ does not mark software functionality, which is concerned with (only) read operations.
The clear specification of read and write responsibilities for data is a prerequisite for the creation of user roles that are used in an authentication matrix, see for example Table F.4.

13.2.5. Quality attributes

As Figure 13.1 shows, this dissertation deals with the delivery of software by focusing on the satisfaction of its functional user requirements. The intention is to aid in the identification of work units to be "done" for implementing desired software functionality by use of patterns. The project progress and the respective delivery success are measured on the basis of the fulfillment of each "done" unit of software development work.

Quality attributes such as usability, security, performance, or maintainability are not yet explicitly addressed or are part of problem-based project planning. These require additional considerations when planning the satisfaction of user needs.

Project management and software development must constantly reassure that the user is satisfied with the product, because this determines the success of the project. This reassurance not only serves the validation that the project is on track by answering the question "Do we build the right thing?[8]", but it also verifies the question of "Do we build the thing right?[9]" due to its strengthening of confidence in the product's use. Software that is used indicates best its fitness to purpose, i.e. it confirms the delivery of a product that satisfies (what) the user (really) needs.

The earlier in the project process this reassurance is made possible, the better user satisfaction[10] can be achieved, one which acknowledges technical excellence and enables the team to "embrace change" at a sustainable pace, i.e. by continuously proceed[11] in the delivery of software that fits its purpose. This is made possible by "A Continual Emphasis on Design" [206, chapter 6] from the beginning of a project, which makes a preview of the expectable quality and its associated work to be "done" available to the project team (users and developers).

The examples paragraph gives a brief answer to the following question of future research:

- *How does* ***Quality-by-Design*** *as a principle of Sustainable Software Development integrate with Problem-Based Project Planning?*

EXAMPLE 13.6 Authorization Conception and User-Centered Design

The concern that "models may be created [. . .] but these models are not maintained and not kept consistent through the further development" [153, page 422] holds for functional user requirements as well as for quality requirements irrespective of an agile or waterfall-like project process execution. As discussed in Section 13.2.2, it depends on a models range of uses and benefits, which would encourage the team to keep it up-to-date and take care of a living documentation. The more critical problem is that quality considerations are too often skipped and not built into the product with the same priority and determination as software features.
For example, security considerations in the context of General Data Protection Regulation [170] require "documentation and privacy analysis tasks" [153, page 422], which are needed in the beginning of development activities and can be hardly built into a product after its delivery. For instance read and write rights on data, such as proposed by Table F.2 List of TF-keywords for problem-based user story templates must be known (be)for(e) the establishment of an Authentication matrix, such as proposed in Table F.5 Permissions matrix for the Vacation Rentals Web Application and Table F.6 Permissions matrix for the Student Recruitment Web Portal for the case studies discussed in this dissertation. Data models and authentication schemas are needed to set up data base configurations in advance to any feature development. Bending these during the development of a software product is usually an expensive, if not futile, endeavour.

[8] Do we meet the actual problem?
[9] Do we meet the purpose by the solution?
[10] In terms of 'built to work', contrary to 'built to last', or 'built to flip' [62, 63], who understands flip as 'through away'
[11] over the long term

EXAMPLE 13.7 Authorization Conception and User-Centered Design, continued

Usability and User Experience is decided at the user interface (UI). That is why each requirements work package according to Table F.3 consists of user story and related acceptance criteria, and a visual representation of the intented UI layout (mockup). A picture is worth a thousand words, and can help achieve consensus within the team about what quality they want to achieve.
Analysis of personas, the development of a product roadmap, the design of the customer journey, can be related to the Storyboard, its roles and scenarios, such as proposed in Table F.8 and Table F.7. In this way, the pre-defined units for problem-based project planning are of multi-purpose for different quality attributes, and help to achieve *Quality-by-Design* right from the start of a project.

13.2.6. Testing

As Figure 13.1 shows, this dissertation deals with user satisfaction by the delivery of a "done" work item that fulfills the functional user requirements. The intention is to make the delivery of desired software functionality by use of patterns possible, i.e. to get the work done (in one or the other way).

From a quality assurance perspective, user satisfaction and user acceptance correlate, especially in projects that follow a user-centered design.

It is worthwhile to have "A Working Product at all times/ [by] Valuing Defect Prevention over Defect Detection" [206, chapter 4 and 5], not only for the implemention of Sustainable Software Development and the achievement of technical excellence, but also to collect user feedback[12] in a timely manner.

Ensuring as soon as possible that the user is satisfied with the way the product is working, not only paves the way to the user's approval to accept the product under development, but also opens the way for using this confirmed observation (of what is working for the user or not) for testing.

Part of future research can involve the question of: *How the development and use of*

- *functional test cases,*
- *test data, and*
- *test management*

can take advantage from the pre-defined units of software functionality as proposed for operating problem-based project planning?

[12]As discussed in Section 2.1 The Software Project Triad, lack of user involvement and insufficient requirements not only belong together, they are also among to top reasons why software projects fail, according to The Standish Group [208].

EXAMPLE 13.8 Automated user acceptance testing

The enumerated scenarios of user stories in Table F.8 Story Mapping for the Vacation Rentals Web Application and Table F.7 Story Mapping for the Student Recruitment Web Portal proved useful for

- reflecting the workflow and to decide on the user stories' prioritization as discussed in Section 13.2.3 Process management
- identifying recurring data chunks that adhere to a user story, providing a retrievable starting point for their refinement as discussed in Section 13.2.4 Data modeling and assist the development of their respective read and write access right and according user roles as dicussed in Section 13.2.5 Quality attributes
- risk- (and problem-)based test management.

By assigning user stories to values for a damage class[a] and a frequency class[b], the discussion in the project team about the criticality of each becomes very productive in regard to the need and development of countermeasures, and the decision of test automation and its depth. This information is successfully integratable with test automation tools. In this way, these tables are of multi-purpose in the project, to the users, to the developers (and testers), and for the team members' to work together. These tables, intended for the project planning, guide the entire project execution, during the development testing and in user acceptance testing (UAT). A project team that finds these useful, will make sure of itself that this document is kept up to date.

[a]damage class: the estimated severity caused by a failed user story, e.g. low, intermediate, or high
[b]frequency class: the estimated occurence of the execution a user story, e.g. infrequent, or frequent

Part VII.

Appendices

Last but not least, Part VII Appendices provides supplementary materials to this dissertation. Appendix A ISO/IEC 20926:2009 Complexity and Size Tables belongs to the input documents of the Frame Counting Agenda, specifying the complexity parameters and point values that can be assigned to a Requirements Work Package. Appendix B Sanity Checks intends to justify the quality and fitness of the requirements sizing method proposed by the frame counting agenda to the standard ISO/IEC 20926:2009 and to the certification practices of the International Function Point Users Group. Appendix C Listing of Philosophies represents a loose collection of philosophies around agile project practices. Appendix D Overview on Architecture Design Patterns enumerates commonly known patterns applicable to software architecture design. Appendix E Structures of Architecture Design Patterns lists the structure of those architecture design patterns, which are discussed in chapter 7.3 Transition Templates – Making problems absorb into platform for the development of solution templates. Appendix F For Further Discussion presents a collection of notes on the further development of the contributions in this dissertation. A List of Tables, List of Figures, and List of Examples are complemented by an overview of Acronyms frequently applied in this dissertation. Finally, a bibliography comprising all References used to this dissertation are offered to its dear reader.

A. ISO/IEC 20926:2009 Complexity and Size Tables

A.1. Data Function Complexity Matrix

		DETs		
		1-19	20-50	>50
RETs	1	Low	Low	Average
	2-5	Low	Average	High
	>5	Average	High	High

TABLE A.1 Data function complexity matrix, taken from [117, table A.1, page 23]

A.2. Data Function Size Matrix

		Type	
		ILF	EIF
Functional Complexity	Low	7	5
	Average	10	7
	High	15	10

TABLE A.2 Data function size matrix, taken from [117, table A.2, page 23]

A.3. Transactional Function Complexity Matrix

		DETs		
		1-4	5-15	>15
FTRs	0-1	Low	Low	Average
	2	Low	Average	High
	>2	Average	High	High

TABLE A.3 EI functional complexity matrix, taken from [117, table A.3, page 23]

		DETs		
		1-5	6-19	>19
FTRs	0-1	Low	Low	Average
	2-3	Low	Average	High
	>3	Average	High	High
NOTE: An EQ has a minimum of 1 FTR.				

TABLE A.4 EO and EQ functional complexity matrix, taken from [117, table A.4, page 23]

A.4. Transactional Function Size Matrix

		Type		
		EI	EO	EQ
Functional Complexity	Low	3	4	3
	Average	4	5	4
	High	6	7	6

TABLE A.5 Transactional function size matrix, taken from [117, table A.5, page 23]

B. Sanity Checks

This appendix provides some quick-and-dirty evaluation for some of the concepts and methods designed in this dissertation. It shows the rationale behind the conceptualization done in this work and provides a means of plausibility check to the reader.

Section B.1 FCA Validation Conditions and the IFPUG Measurement Process gives a mapping of the functional size measurement method taken from ISO 20926 [117, pages 8–22, chapter 5] to the activities and steps of the Frame Counting Agenda as summarized in table 6.2 on page 82 and its Validation Conditions as presented by table 6.3 on page 86ff.

It visualizes the coverage of both approaches to early function point counting, and justifies the deduction of each validation condition by commenting *inline* on these, and giving explicit reference to respective sections in the ISO 20926 standard.

Section B.2 UML4PF and the Criteria for Certification of Function Point Software type 2 investigates which impediments to overcome for making the UML4PF tool [107] an ISO 20926-conform, IFPUG-certifiable software, that assists the requirements engineer or estimator in an automated execution of early functional size measurement.

It compares the fundamentals of problem-based functional size measurement patterns, i.e. the constraints (in table 5.1 on page 48) and criteria (in Definition **DEFINITION 5.5** on page 66) for implementing a set of unique software functionality, as well as the validation conditions of the frame counting agenda with the criteria given for certification of function point software type 2 [121], which allow for certification for “Software [that] provides Function Point data collection and calculation functionality, where the user and the system/software determine the Function Point count interactively. [That is,] The user answers the questions presented by the system/software and the system/software makes decisions about the count, records it and performs the appropriate calculations.” In doing so, a reverse conclusion is possible by which a certifiable UML4PF tool, one that implements problem-based functional size measurement, is a confirmation for the applicability and quality of the means as developed in this dissertation.

B.1. FCA Validation Conditions and the IFPUG Measurement Process

This section gives a mapping of the problem-based functional size measurement method as represented by the Frame Counting Agenda and its involved Validation Conditions to the IFPUG functional size measurement process as documented by ISO 20926 [117, pages 8–22, chapter 5].

The first column, shows the activities and steps of the frame counting agenda, and in its third column it shows the activities of ISO 20926 measurement process. That way, in each row of table B.1 it is illustrated, which activities in either the agenda or the ISO standard method conform to each other. For instance, the *Activity 2. Determine Data Functions.* of the frame counting agenda relates to the activity *5.4 Measure data functions* of ISO 20926. Those rows, which relate to activities, are highlighted in gray color for the sake of readability. Darkgray-colored rows indicate activities, which are of no relevance for the mapping. A discussion about the appropriateness of this deliberate omission is present in section 6.2.2 IFPUG FSM Method ISO/IEC 20926:2009 – Measurement Process on page 78.

In its second column, table B.1 shows the respective validation conditions, which belong to each activity or step of the frame counting agenda. These represent customizations of the counting rules taken from ISO 20926 [117] for using these in functional size measurement by means of patterns, i.e. those as in Table 5.5 on page 72. That is why, the fourth column of table B.1 gives all the references to the ISO standard, which have been of relevance for deducing respective validation conditions. In addition, besides each validation condition an *inline* comment is present, which provides further insights on the reasoning behind creating these.

On a quick look, it becomes obvious that each relevant activity of the IFPUG ISO 20926 standard measurement process is covered by the frame counting agenda in regard to purpose and sequence. ISO 20926 activity *5.4.1 Overview* is not explicitly addressed, since it provides only an overview of subsequent activities and no independent tasks to perform. This is in contrast to activity *5.5.1 Overview*, which is explicitly considered by the frame counting agenda. That is due to the reason, that by start of the activities for *5.5*, only transactional functions are in the focus of considerations. The activities previous to *5.5* focus on data functions. Transactional and data functions are represented by different domains according to the meta model in figure 5.5 on page 47 and its constraints on requirements modeling and counting as presented by table 5.1 on page 48. That is, activity *3.a Identify machine domain as transactional function.* is intended for making the estimator aware that there is a change regarding the elements, which undergo the functional size measurement. It can be discussed, if the actitivities *3.a* and *3.b* of the frame counting agenda should be merged. *Activity 4. Report Functional Size for FUR.* covers two activities of the ISO 20926 measurement process, namely *5.9 Document the function point count* and *5.10 Report the results of the function point count*. This is reasonable, since *5.9.* summarizes which documents must be available for reporting the result of a function point count, and activity *5.10* only demands from an ISO 20926-conform functional size measurement to document and indicate any costumizations by a respective postfix to the result of a function point count. That is why it is reasonable to merge the activities *5.9* and *5.10* into one.

Comments on origin and meaning for each validation condition can be found *inline* to the following explanations of table B.1.

Frame Counting Agenda	Validation Conditions	ISO 20926 Measurement Process, taken from [117, pages 8–22, chapter 5]	ISO 20926 reference
		5.1 Overview	[117, pages 8–9, section 5.1]
		5.2 Gather the available documentation	[117, page 9, section 5.2]

Comment: For initiating the function point count, the base functional components (ILF, EIF, EI, EQ, EO) must be identified. This activity is assisted by and reduced to the consideration of patterns for requirements in problem-based functional size measurement. In this case, functional size measurement (FSM) starts with classifying functional user requirements (FUR) by means of specific problem-based functional size measurement patterns, which is described next.

Activity 1. Classify FUR by FSM Patterns.		5.3 Determine the counting scope and boundary and identify FUR	[117, pages 9–10, section 5.3]

Comment: According to ISO 20926, this activity serves the identification of the a) purpose of the count, b) type of count, c) counting scope, d) boundary of application within counting scope based on user view, e) functional requirements, not the non-functional ones. These concerns are addressed as follows: a) The purpose of count is fix, since the purpose is to determine the functional size of a requirements work package. b) The type of count is fix, too. There is no type of count, since different calculations for determining the functional size of a requirements work package do not exist and are not required. c) The counting scope, it is limited to one problem given in a requirements work package. d) The application boundary is based on the user view and reflected by the machine interface. The patterns used care for a problem-oriented perspective on the requirements, such that technical considerations are excluded from considerations by design. e) The problem-based functional size measurement patterns as in table 5.5 Basic Problem Frames with relevance in Functional Size Measurement on page 72 focus on functional requirements.

$\left.\begin{array}{l}\textbf{V.i}\\ \textbf{V.ii}\end{array}\right\}$ *take care that only one, independent problem is going to be counted.*

V.iii *assures that functional size measurement takes place at a defined application boundary, which in this work equates with the machine interface, cf. table 5.1 on page 48.*

Activity 2. Determine Data Functions.		5.4 Measure data functions	[117, pages 10–13, section 5.4]
		5.4.1 Overview	[117, page 10, section 5.4.1]

Comment: This activity is subdivided into the steps 2.a to 2.f in the frame counting agenda as discussed next.

Continued on next page...

Frame Counting Agenda	Validation Conditions	ISO 20926 Measurement Process, taken from [117, pages 8–22, chapter 5]	ISO 20926 reference
2.a Identify problem domains as data functions.		5.4.2 Identify and group all logical data into data functions	[117, pages 10–11, section 5.4.2]

Comment: This activity serves according to ISO 20926 for identifying those a) logical groups of c) related and user recognizable data information, which are involved with the requirements under consideration. Thereby, keeping the user perspective on these data is of first importance. Code data or d)-f) data specially created for implementing a particular solution as for instance data base foreign keys, etc., or data, which b) is out of scope for the problem at hand, must be clearly excluded.
These groups of logically related data is associated with the concept of problem domains in the meta-model as presented in figure 5.5 on page 47 and detailed by table 5.1 on page 48, which are fundamental to each problem-based functional size measurement pattern. That is, the grouping of logical related data comes by design when using these patterns for building the requirements model. In addition, these patterns focus the problem by making explicit reference to the user perspective as given in the requirements. All parts and details to any possible solution are hidden in the machine black box, and thus not considered in this representation of the requirements.

V.iv *identifies all problem domains, which are candidates for being evaluated as data function.*

V.v *excludes all problem domains from becoming data functions, which share only causal phenomena and thus control information with the machine.*

Frame Counting Agenda	Validation Conditions	ISO 20926 Measurement Process, taken from [117, pages 8–22, chapter 5]	ISO 20926 reference
2.b Classify data functions into ILF or EIF.		5.4.3 Classify each data function as either ILF or EIF	[117, page 11, section 5.4.3]

Comment: This activity is a refinement of the results provided by the previous one, where two types of data functions must be distinguished according to ISO 20926 in a) Internal Logical Files (ILF) and b) External Interface Files (EIF). Since problem domains are either of a causal (C), biddable (B), lexical (X) or display (D) domain type, these characteristics assists the separation of data functions as either ILF or EIF. Note: The question to answer here is, what type of problem domain, that is in control of symbolic phenomena at the machine interface, is an ILF and which one is an EIF.

V.vi *is concerned with symbolic phenomena that are controlled by the machine domain. This VC expresses that respective problem domains, i.e. the ones whose symbolic phenomena are controlled by the machine, represents an ILF, since these are "maintained by the application being measured." [117, page 11, section 5.4.3 a)].*

Continued on next page...

Frame Counting Agenda	Validation Conditions	ISO 20926 Measurement Process, taken from [117, pages 8–22, chapter 5]	ISO 20926 reference
	V.vii *is concerned with symbolic phenomena that are controlled by problem domains. These domains represent data groups, which are "referenced, but not maintained by the application being measured [. . . and may be] identified in an ILF in one or more other applications" [117, page 11, section 5.4.3 b)]. Respectively, a data function cannot take the role of an ILF and an EIF simultaneously. For all four types of problem domains it must be decided, if it is counted as in ILF or as an EIF. This is implemented by the following validation conditions:* ***V.viii***, ***V.ix***, *and* ***V.x*** *as follows.*		
	V.viii *allows for the most generic domain type of a causal (C) problem domain to be either EIF or ILF. It must be only assured that across all counts of this domain, its role as EIF or ILF is maintained consistently.*		
	V.ix *limits biddable domains to represent EIF only, due to the fact, that a biddable domain type cannot be placed with a requirements constraint according the Jackson's original problem frames approach [128]. Thus, a biddable (B) problem domain can never be an ILF. There are no phenomena controlled by the machine, which are shared at a firsthand interface to any biddable domain.*		
	V.x *makes allowance for the fact, that problem domains of a lexical (X) or display (D) type represent passive ones (see [68, table 1]), which would never create data by themselves. They represent data sinks, which are accessible for the machine domain only. They take the role of ILFs in functional size measurement exclusively, since the machine domain uses these for storing or representing data information. That is, each time a machine domain controls symbolic phenomena at an interface to a lexical or display domain, it manages data, which is internal to the software application under consideration, or at least processed to some extend by it.*		
2.c Count DET for each data function.		5.4.4 Count DETs for each data function	[117, page 11, section 5.4.4]
Comment: After having clarified which problem domain (type C, B, X, D) represents what kind of data function (ILF, EIF) in the previous activity 2.b, ISO 20926 requires to count the number of data element types (DET) for each data function. In doing so, special care must be taken to avoid double counts of DET, which represent "unique user recognizable, non-repeatable attributes maintained in or retrieved from the data function" [117, page 11, section 5.4.4 a)] or respective symbolic, shared phenomena at the interface of problem domains and the machine.			

Continued on next page. . .

Frame Counting Agenda	Validation Conditions	ISO 20926 Measurement Process, taken from [117, pages 8–22, chapter 5]	ISO 20926 reference
For each of these, count a) one DET, and make sure that only those attributes that are b) used and referenced by the software application under count, c) checked for relationship with another data function, and d) reasonably grouped in relation to an elementary process, are taken into considerations. *Each of these points a) to d) as postulated by ISO 20926 in its activity 5.4.4 is supported by design in problem-based functional size measurement. By means of problem-based functional size measurement patterns d) each unit of measure, i.e. a requirements work package is tailored to exactly one elementary process, which b) involves a defined scope of problem context, namely only those problem domains, which are of relevance to the requirements. In addition, c) in each requirement work package, it is the machine domain, which represents the application boundary and establishes relationships between data functions. Its interface allows for analyzing respective dependencies and potential duplicated appearances of shared phenomena in and across requirements work packages. Maybe multiple counts of the same DET are adjusted in activity 2.e of the frame counting agenda, when all necessary information for determining the functional size of one requirements work package is collated.* *Note: Causal phenomena are of relevance to transactional functions. These are counted in activity 3. of the frame counting agenda as discussed later.*			
	V.xi *ensures that determining the functional size of data functions is related to symbolic phenomena only.*		
2.d Count RET for each data function.		5.4.5 Count RETs for each data function	[117, page 12, section 5.4.5]
Comment: In ISO 20926 activity 5.4.5 serves the counting of record element types (RET), which are "user recognizable sub-group of data element types within a data function" [117, page 7, section 3.46]. Problem domains relate to data functions. Both represent a logical grouping of shared phenomena or respective DET by definition. In accordance with IFPUG counting practices [118, section 6-9, RET rules] and ISO 20926 activity 5.4.5 b) the consideration of additional logical sub-grouping of DETs within one data function is neglectable, and thus 5.4.5 a) one RET is counted for each data function by default.			
	V.xii *takes advantage of the structuring for requirements as provided by problem-based functional size measurement patterns. These patterns allow for simple identification and respective counting of RETs for each data function in a requirements work package.*		
2.e Determine functional complexity for data functions.		5.4.6 Determine the functional complexity for each data function	[117, page 12, section 5.4.6]
Comment: This activity takes the Data function complexity matrix, taken from [117, table A.1, page 23] into account for determining the functional complexity of a requirements work package based on its data functions.			

Continued on next page...

Frame Counting Agenda	Validation Conditions	ISO 20926 Measurement Process, taken from [117, pages 8–22, chapter 5]	ISO 20926 reference

In contrast to ISO 20926 activity 5.4.6, which determines the functional complexity for each identified data function in isolation based solely on the Data function complexity matrix, taken from [117, table A.1, page 23], an approach which comes along with the risk of malcounts due to the unawareness of data function relationships and with it the risk of accidently missing the repeated occurence of a DET, the accordant activity 2.e in the frame counting agenda benefits from the grouping of data functions, i.e. the meaningful relationship of problem domains and their shared phenomena as represented by a requirements work package for the application of ISO 20926 data function complexity table. It is build on patterns, which make a systematic analysis of data functions and their DETs possible. It is guided by the validation conditions as follows for adjusting the results of functional size measurement, i.e. counted RET and DET for all data functions in a requirements work package, as have been determined so far.
Note: The intended of problem-based functional size measurement is to enable consistent requirement estimates, which are represented by reproducible function points. Therefore, it applies patterns as a requirements model for executing the function point count. ISO 20926 makes no constraints on what requirements model is to use. That is, the following validation conditions represent an interpretation of ISO 20926 activity 5.4.6, such that it is implementable by problem-based functional size measurement. Have in mind, that in the sense of Carveth Read: it is better to be approximately right than exactly wrong [187].

V.xiii *summarizes the count of DET for all ILF in a requirements work package (RWP) to one cumulated value. This is reasonable, since within one RWP it is in the responsibility of the machine domain to take care for the symbolic phenomena counted as DET of all ILF, i.e. in either requesting or preparing these by its interfaces with respective problem domains. A data function classified as ILF is usually a problem domain that takes the role of a data source, which is internal(ly accessible) to (and thus inside the boundary of) the software application under consideration.*

V.xiv *summarizes the count of DET for all EIF in a requirements work package to one cumulated value. This is reasonable, since the phenomena counted as DET at the machine interface of an EIF are under the control of the problem domains that belong to one RWP. A data function classified as EIF usually represents a(n interface to a) data source external to the software application (boundary) under consideration, e.g. such as a user, or an other software application, which are comparably defined in ISO 20926.*

Continued on next page...

Frame Counting Agenda	Validation Conditions	ISO 20926 Measurement Process, taken from [117, pages 8–22, chapter 5]	ISO 20926 reference
	V.xv *benefits from the logical grouping of data functions in one requirements work package. It adjusts the count of DET for all ILF and EIF in regard to the multiple occurence of shared phenomena, whose repeated consideration within one function point count must be strictly avoided and justified. For instance, cf. the simple workpieces problem as discussed in detail in section Step-By-Step Guide to the Requirements Sizing Method. In this problem, the problem domain "User" (EIF) controls the symbolic phenomenon "Guest" ($= 1$ DET), and the "Party Plan" (ILF) controls the same symbolic phenomenon "Guest" ($= 1$ DET) on behalf of the machine domain, too. In order to avoid accidently double counts of here the "Guest" phenomenon (as $\neq 2$ DET) for this requirements work package, this validation condition makes clear that counting the DET of an ILF suffices in this case, since what happens at this machine interfaces and respective problem domain "Party Plan" is constrained by the (functional user) requirements, and thus makes the functionality to be delivered by the software application to-be. That is why, after application of this validation condition, DET_{ILF} remains one (1 DET) and DET_{EIF} becomes zero (0 DET) for the "Guest" phenomenon in this requirements work package.*		
	V.xvi *considers the case, where two different data functions classified as ILF in one requirements work package, are involved with the same DET. 9 out of 17 patterns for functional size measurement, i.e. frames no. #01, #03–05, #12, #13, #15–17 as in table 5.5 on page 5.5, are candidate problems for this case, which illustrates the importance of this validation condition. For instance, the functional size measurement patterns* ***#03 PF 3.2*** *and* ***#04 PF 3.4*** *represent transformation problems, which simply move data information from one place or lexical problem domain (X) to the next (X) by means of an external input (EI) process (TOFF-i. functionality). Remember that a lexical domain (X) can only take the role of an ILF in problem-based functional size measurement, since these are internal to the software application (boundary) according to validation condition* ***V.x****. This means, that it is finally the machine, which holds the responsibility for maintaining their DET. This situation comes with two implications: First, for each ILF referenced in one problem, there is (or should be) another problem available, where this referenced ILF is a constrained one. Second, double counts of the same DET in one requirements work package are (easily) preventable by counting only those of the constrained problem domain.* *Note: As illstrated by this validation condition, problem-based functional size measurement assists not only the consistency of function point counts, but also enables a second field of application to the analyst or estimator, namely that of checking the requirement's completeness with respect to a specific problem domain. By use of patterns the analysis of inevitable problem compositions or work package dependencies can be supported, such as e.g. each time a "simple workpieces" problem is identified, which is intended for making some data information persistent, a "display information" problem is also involved for providing feedback on the status of storing these data, etc. This approach prevents from overseeing the "gray rhino" requirements, which are the "obvious thing that's coming at you" [205].*		

Continued on next page…

Frame Counting Agenda	Validation Conditions	ISO 20926 Measurement Process, taken from [117, pages 8–22, chapter 5]	ISO 20926 reference
	V.xvii – **V.xviii** } *differentiate between ILF (constrained domain with symbolic phenomena) and EIF (referenced domain with symbolic phenomena) for determining in accordance with validation condition **V.xii** the total number of problem domains or their respective machine interfaces (RET) in one requirements work package.*		
	V.xix *takes ILF and EIF specific RET from validation conditions **V.xvii** and **V.xviii** above, as well as DET from validation conditions **V.xiii** to **V.xvi**, and applies these as input to ISO 20926 table A.1 Data function complexity matrix, taken from [117, table A.1, page 23]. In this context, validation condition **V.xix** covers the case, where one of these values is zero (0), for instance due to multiple occurences of some DET. Then, {n/a} which stands for "not applicable" becomes a kind of default value for their respective functional complexity, which allows for continuing in the frame counting agenda.*		
	V.xx *determines the functional complexity for all data function of one requirements work package, which are classified as ILF.*		
	V.xxi *determines the functional complexity for all data function of one requirements work package, which are classified as EIF.*		
2.f Determine functional size for data functions.		5.4.7 Determine the functional size for each data function	[117, page 13, section 5.4.7]
Comment: This activity takes the Data function size matrix, taken from [117, table A.2, page 23] into account for determining the functional size of a requirements work package. After completing this activity 2.f of the frame counting agenda, the size for all data functions within one requirements work package is available given by a numerical value in function points.			
	V.xxii *considers the case, where according to **V.xix** in activity 2.e determining a functional complexity is not possible. It sets the size of respective data functions to zero (0) function points by default.*		
	V.xxiii – **V.xxiv** } *consider the case, where the functional complexity for some data functions is available. These validation conditions ensure that respective functional sizes are determined for all ILF and EIF within one requirements work package. Note: Placing constraints on what problem domain type (X, D, C, or B) belongs to what data function (ILF or EIF) as done by the validation conditions **V.vi** to **V.x** before, is now to the advantage of reproducible function points. The input to table Data function size matrix, taken from [117, table A.2, page 23] depends on these constraints. Thus, its systematic application is facilitated by the use of problem-based functional size measurement patterns and by the validation conditions as part of the frame counting agenda.*		

Continued on next page...

Frame Counting Agenda	Validation Conditions	ISO 20926 Measurement Process, taken from [117, pages 8–22, chapter 5]	ISO 20926 reference
Activity 3. Determine Transactional Function.		5.5 Measure transactional functions	[117, page 13, section 5.5]
3.a Identify machine domain as transactional function.		5.5.1 Overview	[117, page 13, section 5.5.1]
3.b Classify transactional function as either EI, EQ, or EO.		5.5.2 Identify elementary processes	[117, pages 13–15, section 5.5.2]
		5.5.3 Classify each elementary process as a transactional function	[117, page 16, section 5.5.3]

*Comment: This activity 3. Determine Transactional Function is subdivided into the steps 3.a to 3.f by the frame counting agenda. It integrates the activities 5.5.2 and 5.5.3 of the ISO 20926 functional size measurement process to 3.b, since elementary processes and transactional functions become synonymic by the application of problem-based functional size measurement pattern. In general, activity 3. of the frame counting agenda is concerned with determining the functional size of transactional functions, e.g. the interactions required for moving data information from source to sink or for providing some control information for triggering actions. So, the main interest here is to identify the form of processing data and to count the interfaces within one requirements work package. According to ISO 20926 activity 5.5.1 this should be done for "all transactional functions within the counting scope" [117, page 13, section 5.5.1], of which in a requirements work package exactly one exists, which is represented by the machine domain. Problem-based functional size measurement patterns are tailored to produce problems, which represent one specific transactional function aka elementary process (EI, EQ, EO), cf. table 5.5 on page 72. Thus, their use realizes ISO 20926 activities 5.5.2 and 5.5.3 by design. Maybe the following validation condition **V.xxv** is be perceived as expendable, but it is left in for creating awareness of what to measure next.*

> **V.xxv** *marks the shift in perspective on what counts in the measurement procedure. It clearly indicates to the estimator, that transactional functions are in the focus of considerations now, and not the data functions. In addition, it ensures that problem-based functional size measurement patterns are used for executing the function point count. Otherwise, consistent requirements estimates cannot be expected by use of the frame counting agenda.*

Note: Activities 5.2.2 and 5.2.3 in ISO 20926 functional size measurement process provide a huge catalog of hints and examples, i.e. heuristics for identifying and classifying transactional functions. Their use becomes obsolete by the application of problem-based functional size measurement patterns, which save the estimator much inspection effort and limits the risk of misinterpretation regarding these heuristics. Nevertheless, subsequent paragraphs comment on the subactivities 5.5.2.1 to 5.5.2.3, and 5.5.3.1 to 5.5.3.2 in some brief statements.

Continued on next page...

Frame Counting Agenda	Validation Conditions	ISO 20926 Measurement Process, taken from [117, pages 8–22, chapter 5]	ISO 20926 reference

ISO 20926 activity 5.5.2.1 for identifying elementary processes can be fulfilled, since problem-based functional size measurement patterns are design for a) composing and decomposing the functional user requirements into units of measure for requirements, which b) at least create a.2) a complete, and a.1) a meaningful unit for requirements estimating. This is established by ***DEFINITION 5.1*** *for Problem Unit – Requirements Work Package in section 5.3 on page 46.*

ISO 20926 activity 5.5.2.2 a), b) are covered by problem-based functional size measurement, since a) similar elementary processes are identifiable based on the patterns applied, cf. table 5.5 Basic Problem Frames with relevance in Functional Size Measurement , and their involved type of functionality as introduced by ***DEFINITION 5.3****, and b) multiple forms of finally the same elementary process are packaged to one requirements work package by following the criteria in* ***DEFINITION 5.1****.*

ISO 20926 activity 5.5.2.3 lists in its table 3 [117, page 14] thirteen examples for different forms of processing logic. Each of these is reducible to one elementary process (EI, EO, EQ) and thus representable by problem-based functional size measurement patterns as in table 5.5.

ISO 20926 activity 5.5.3.1 demands to identify the primary intent for each elementary process by some characteristics a) to c), which conforms to particular types of problem domains as discussed in section 5.4 and summarized by table 5.2, and thus are inherent to problem-based functional size measurement patterns.

ISO 20926 activity 5.5.3.2. and its definition of a) EI, b) EO, c) EQ, have been strengthened to what has been already explained in the paragraphs above. Its table 4 – relationship between primary intent and transactional function types has inspired table 5.2 Mapping problem frames to elementary processes by types of functionality, and table 5 – relationship between processing logic and transactional function type induced the idea for table 5.5 Basic Problem Frames with relevance in Functional Size Measurement .

3.c Count FTR for transactional function.		5.5.4 Count FTR for each transactional function	[117, page 17, section 5.5.4]

Comment: ISO 20926 activity 5.5.4 demands to count one (1) FTR for each data function accessed by the transactional function.

V.xxvi *correlates the count of FTR with the number of interfaces at the machine domain. In one requirements work package, this conforms to the total number of problem domains that share an interface with the machine.*

Continued on next page...

Frame Counting Agenda	Validation Conditions	ISO 20926 Measurement Process, taken from [117, pages 8–22, chapter 5]	ISO 20926 reference
3.d Count DET for transactional function.		5.5.5 Count DETs for each transactional function	[117, page 17, section 5.5.5]

Comment: ISO 20926 activity 5.5.5 demands to count b),[d)] unique attributes as DETs for a transactional function, which a) cross the application boundary, whereby making explicit that c), e) certain kinds of DET, which do not fulfill 5.5.5 a, b),[d)] are excluded from the count, such as those which reside inside the application boundary only, or DETs, which are only present due to technical reasons, but not related to the functional user requirements.
Deciding on the location of DETs in regard to the application boundary is eased by problem-based functional size measurement, since these take the type of involved problem domains into account.

V.xxvii *makes allowance for the fact that counting DET for data functions or transactional functions serves different purposes in functional size measurement. So each shared phenomenon at the machine interface is again part of considerations. In case of transactional functions, symbolic as well as causal phenomena are of relevance.*

V.xxviii *takes into account that a lexical (X) problem domain relates to an Internal Logical File (ILF), cf. validation condition* ***V.x****, which resides inside the application boundary and thus does not count as DET for a transactional function, whose DETs have to cross the application boundary. Note: In order to avoid double counts of DET, shared phenomena at the machine interface to a lexical problem domain do not count for transactional functions. First, they have been already considered as data functions. Second, for each machine interface to a lexical domain there exists a machine interface to a biddable or a causal problem domain in the same requirements work package, which holds corresponding shared phenomena. For instance, in a simple workpieces problem such as discussed in section 6.4 Step-By-Step Guide to the Requirements Sizing Method, a (biddable) user (domain) has control of phenomena for planning a list of guests for her party, e.g. User!{plan, Guest}. These phenomena are also present at the machine interface to the respective (lexical) party plan (domain), and have been already counted as data function. Considering these phenomena in the count of a transactional function would yield a replicated and thus to a malcount of these.*

V.xxix *takes into account that causal (C) and biddable (B) problem domains in general relate to External Interface Files (EIF), cf. validation conditions* ***V.viii*** *and* ***V.ix****, which reside outside the application boundary. Their interaction with the machine domain, which represents the transactional function in a requirements work package, requires their DET to cross the application boundary. The display (D) problem domain type is often in use as a kind of connecting domain between the machine and a biddable problem domain, which requires respective DET also to cross the application boundary (as seen from the requirements perspective on this problem).*

Continued on next page...

Frame Counting Agenda	Validation Conditions	ISO 20926 Measurement Process, taken from [117, pages 8–22, chapter 5]	ISO 20926 reference
	V.xxx } **V.xxxi**	*After identifying those shared phenomena, which cross the application boundary by validation condition* ***V.xxvii*** *to* ***V.xxix****, each of these counts one (1) DET for the transactional function in this requirements work package.*	
3.e Determine functional complexity for transactional function.		5.5.6 Determine the functional complexity for each transactional function	[117, page 18, section 5.5.6]
Comment: This activity takes the EI functional complexity matrix, taken from [117, table A.3, page 23] as well as the EO and EQ functional complexity matrix, taken from [117, table A.4, page 23] into account for determining the functional complexity of a requirements work package for its transactional function. The functional complexity of a transaction function ($TF_{Complexity}$) depends on its type (TF_{Type}) as determined in activity 3.b of the frame counting agenda, and its belonging number of FTR and DET as determined in activity 3.c and 3.d.			
	V.xxxii	*TF_{Type} defines what functional complexity Table A.3 or Table A.4 is to use for determining the functional complexity of the transactional function. TF_{FTR} and TF_{DET} provide input values to both tables, such that a functional complexity of either Low, Average, or High becomes ascertainable.*	
3.f Determine functional size for transactional function.		5.5.7 Determine the functional size of each transactional function	[117, page 19, section 5.5.7]
Comment: This activity takes the Transactional function size matrix, taken from [117, table A.5, page 23] into account for determining the functional size of the transactional function within one requirements work package. After completing this activity 3.f of the frame counting agenda, the size for the transactional function is available given by a numerical value in function points.			
	V.xxxiii	*TF_{Type} as determined in activity 3.b of the frame counting agenda, and $TF_{Complexity}$ as determined in the previous activity 3.e, represent input to Table A.5 for ascertaining the functional size of the transactional function.*	
		5.6 Measure conversion functionality	[117, page 19, section 5.6]
		5.7 Measure enhancement functionality	[117, page 19, section 5.7]
		5.8 Calculate functional size	[117, pages 19–20, section 5.8]
Comment: ISO 20926 activities 5.6 to 5.8 are not addressed by the frame counting agenda as motivated in section 6.2.2.			

Continued on next page...

Frame Counting Agenda	Validation Conditions	ISO 20926 Measurement Process, taken from [117, pages 8–22, chapter 5]	ISO 20926 reference
Activity 4. Report Functional Size for FUR.		5.9 Document the function point count	[117, page 21, section 5.9]
		5.10 Report the results of the function point count	[117, pages 21–22, section 5.10]
Comment: ISO 20926 activity 5.9 provides a list of information, some of them are mandatory and some of them are optional, that have to be documented for a function point count. This list is fulfilled by the use of a requirements work package for executing the functional size measurement process, as it comprises a requirements model, which is in accordance with this standard. Due to the application of problem-based functional size measurement patterns, a requirements work package makes all the attributes transparent and documentable that have to be counted for measuring the functional size of this unit for requirements. According to ISO 20926 5.10.2 the results of the frame counting agenda, which implements a problem-based functional size measurement process, represent a customization of the measurement method as defined in the standard. Respectively, its results shall be reportet as: S FP (IFPUG-IS-FCA), where S is the number of function points, FP is the unit of size in functional size measurement, namely function points, IS indicates the application of ISO 20926 standard (ISO/IEC 20926:2009) [117], and FCA marks the utilization of the frame counting agenda for executing the measurement process. At the end of this activity 4., which also marks the end of the functional size measurement for one requirements work package, there is one function point value, which represents the functional size of its involved requirements.			
	V.xxxiv *The number of function points S for a requirements work package is a cumulated one, based on the functional size for the data functions (ILF, EIF) as determined in activity 2.f, and the functional size for the transactional function as determined in activity 3.f of the frame counting agenda.*		

TABLE B.1 Sanity check of problem-based functional size measurement by ISO 20926 [117]

B.2. UML4PF and the Criteria for Certification of Function Point Software type 2

General idea: UML4PF [107] is an eclipse plugin for modeling requirements based on the problem frames approach by Jackson [128]. An extension of this plugin by problem-based functional size measurement would establish an automated functional size measurement tool, which is in accordance with the IFPUG ISO 20926 functional size measurement method. The IFPUG has published their criteria used for the certification of such a software tool [121]. These criteria are taken for a quick plausibility check of problem-based functional size measurement, and their check serves as a simple feasibility study for the intented UML4PF extension as well.

Results at a glance: While executing this check, two issues arise:

The first issue refers to the criteria 3.e, 4.g, and 5.g for certifying function point software type 2, which all are concerned with assessing the "capability [...] to send a system response [..., which] is counted as one DET" [121]. With regard to a requirements work package this implies to add the lump-sum of 1 DET to its count. This does not affect its functional size significantly, especially if it is done by default for each requirements work package. In summary, this first issue is negligible.

The second issue refers to the criteria 3.f, 4.h, and 5.h for certifying function point software type 2, which all are concerned with assessing the "ability [...] to specify an action to be taken is counted as one DET, even if there are multiple methods for invoking the same logical process". This issue is already addressed by the kind of decomposing requirements to the same work package as established by the criteria given in **DEFINITION 5.1**. It implements the measurement rule 5.5.2.2.b) in ISO 20926 [117, page 14] to "Do not split an elementary process with multiple forms of processing logic into multiple elementary processes." For instance, as discussed for the Vacation Rentals Web Application example in the Case Studies chapter, browsing the holiday offers of a vacation rentals web site may be done by a guest or a staff member, but the problem per se, namely "browsing holiday offers" remains the same (logical process). Thus, the proposed requirements sizing method addresses this second issue as well.

Recommendations: Extending the UML4PF tool by the Requirements Sizing Method as proposed in this dissertation is a worthwhile idea for automating the estimation of requirements, and determining a function point value for these, which is in conformance with the IFPUG ISO 20926. It would allow for a software-supported, pattern-based managing and measuring of requirements at once. This extension would be certifiable according to the criteria used for the certification of function point software type 2 [121], which also emphasizes the quality of problem-based functional size measurement as introduced in this dissertation.

C. Listing of Philosophies

C.1. 7 Lean Principles [156]

1. Eliminate waste
2. Amplify learning
3. Decide as late as possible
4. Deliver as fast as possible
5. Empower the team
6. Build integrity in
7. See the whole

C.2. 3 Forms of Waste addressed in Lean Production [148]

Muda:	"futility; uselessness; idleness; **superfluity**; waste; wastefulness" or failures of people or processes to efficiently deliver product.
Mura:	"unevenness; irregularity; **lack of uniformity**; inequality", or failures related to unpredictable or inconsistent outputs.
Muri:	"unreasonableness; impossible; beyond one's power; too difficult; by force; compulsorily; excessive; immoderation, Overburden", or **failures of standardization** to create efficient process.

C.3. 5 CMMI Maturity Levels for services [123]

Maturity Level 5 –	**Optimizing**: Stable and flexible. Organization is focused on continuous improvement and is built to pivot and respond to opportunity and change. The organization's stability provides a platform for agility and innovation.
Maturity Level 4 –	**Managed**: Measured and controlled. Organization is data-driven with quantitative performance improvement objectives that are predictable and align to meet the needs of internal and external stakeholders.
Maturity Level 3 –	**Defined**: Proactive, rather than reactive. Organization-wide standards provide guidance across projects, programs and portfolios.
Maturity Level 2 –	**Managed**: Managed on the project level. Projects are planned, performed, **measured**, and controlled.
Maturity Level 1 –	**Initial**: Unpredictable and reactive. Work gets completed but is often delayed and over budget.

C.4. 12 Agile Software Development Principles behind The Manifesto [30]

1. Customer satisfaction by early and continuous delivery of valuable software.
2. Welcome changing requirements, even in late development.
3. Working software is delivered frequently (weeks rather than months).
4. Close, daily cooperation between business people and developers.
5. Projects are built around motivated individuals, who should be trusted.
6. Face-to-face conversation is the best form of communication (co-location).
7. **Working software is the primary measure of progress.**
8. **Sustainable development, able to maintain a constant pace.**
9. Continuous attention to technical excellence and good design.
10. Simplicity – the art of **maximizing the amount of work not done** – is essential.
11. Best achitectures, requirements, and designs emerge from self-organizing teams.
12. Regularly, the team reflects on how to become more effective, and adjusts accordingly.

C.5. 9 Scaled Agile Framework (SAFe) Lean-Agile Principles [188]

1. Take an economic view
2. Apply systems thinking
3. **Assume variability; preserve options**
4. Build incrementally with fast, integrated learning cycles
5. **Base milestones on objective evaluation of working systems**
6. Visualize and limit WIP, reduce batch size, and manage queue lengths
7. Apply cadence, synchronize with cross-domain planning
8. Unlock the intrinsic motivation of knowledge workers
9. Decentralize decision-making

C.6. 7 Principles of Earned Value Management System [6, 17]

1. Plan all work scope for the program to completion.
2. Break down the program work scope into finite pieces that can be assigned to a responsible person or organization for control of technical, schedule, and cost objectives.
3. Integrate program work scope, schedule, and cost objectives into a performance measurement baseline plan against which accomplishments may be measured. Control changes to the baseline.
4. Use actual costs incurred and recorded in accomplishing the work performed.
5. Objectively assess accomplishments at the work performance level.
6. Analyze significant variances from the plan, forecast impacts, and prepare an estimate at completion based on performance to date and work to be performed.
7. Use this performance information in the organization's management processes.

D. Overview on Architecture Design Patterns

D.1. Architectural Styles

Pattern Name	Pattern Category
Batch Sequential Pipes and Filters	Dataflow Systems
Main Program and subroutine OO systems Hierarchical Layers	Call-and-return Systems
Communicating processes Event systems	Independent Components
Interpreters Rule-Based Systems	Virtual Machines
Databases Hypertext systems Blackboards	Data-centered Systems (repositories)

TABLE D.1 Architectural Styles [198] by Shaw and Garlan (1996)

D.2. Software Architecture

Pattern Name	Pattern Category
Layers	Architectural patterns
Pipes and Filters	
Blackboard	
Broker	
Model–View–Controller	
Presentation-Abstraction-Control	
Microkernel	
Reflection	
Whole-Part	Design patterns
Master-Slave	
Proxy	
Command Processor	
View Handler	
Counted Pointer	
Forwarder–Receiver	
Client-Dispatcher-Server	
Publisher–Subscriber	

TABLE D.2 Software Architecture [46] by Buschmann et al. (1996)

D.3. Design Patterns

Pattern Name	Pattern Category
Abstract Factory	Creational patterns
Builder	
Factory Method	
Object Pool	
Prototype	
Singleton	
Adapter	Structural patterns
Bridge	
Composite	
Decorator	
Facade	
Flyweight	
Private Class Data	
Proxy	
Chain of Responsibility	Behavioral patterns
Command	
Interpreter	
Iterator	
Mediator	
Memento	
Null Object	
Observer	
State	
Strategy	
Template Method	
Visitor	

TABLE D.3 Design Patterns [90] by Gamma et al. (1995)

D.4. Cloud Computing Patterns

Pattern Name	Pattern Category
⋮ Block Storage Blob Storage Relational Database Key-Value Storage Strict Consistency Eventual Consistency	Cloud Offerings/ Storage Offerings
⋮ Stateful Component Stateless Component User Interface Component Processing Component Batch Processing Component Data Access Component Data Abstractor Idempotent Processor Transaction-based Processor Timeout-based Message Processor Multi-Component Image ⋮	Cloud Application Architectures/ Cloud Application Components
Restricted Data Access Component Message Mover Application Component Proxy Compliant Data Replication Integration Provider ⋮	Cloud Application Architectures/ Cloud Integration

TABLE D.4 Cloud Computing Patterns [85] by Fehling et al. (2014)

D.5. Enterprise Integration Patterns

Pattern Name	Pattern Category
File Transfer Shared Database Remote Procedure Invocation Messaging	Integration Styles
Message Channel Message Pipes and Filters Message Router Message Translator Message Endpoint	Messaging Systems
Point-to-Point Channel Publish-Subscribe Channel Datatype Channel Invalid Message Channel Dead Letter Channel Guaranteed Delivery Channel Adapter Messaging Bridge Message Bus	Messaging Channels
Command Message Document Message Event Message Request-Reply Return Address Correlation Identifier Message Sequence Message Expiration Format Indicator	Message Construction
Content-Based Router Message Filter Dynamic Router Recipient List Splitter Aggregator Resequencer Composed Msg. Processor Scatter-Gather Routing Slip Process Manager Message Broker	Message Routing

Pattern Name	Pattern Category
Envelope Wrapper	Message Transformation
Content Enricher	
Content Filter	
Claim Check	
Normalizer	
Canonical Data Model	
Messaging Gateway	Messaging Endpoints
Messaging Mapper	
Transactional Client	
Polling Consumer	
Event-Driven Consumer	
Competing Consumers	
Message Dispatcher	
Selective Consumer	
Durable Subscriber	
Idempotent Receiver	
Service Activator	
Control Bus	System Management
Detour	
Wire Tap	
Message History	
Message Store	
Smart Proxy	
Test Message	
Channel Purger	

TABLE D.5 Enterprise Integration Patterns [112] by Hohpe and Woolf (2003)

E. Structures of Architecture Design Patterns

This appendix lists the architecture design patterns used in sections 7.5.2 through 7.5.5 for developing a new representation out of these, which is referred to as solution templates. Solution templates allow a problem-based functional user requirements model to be assigned to each solution architecture design in a pattern-to-pattern way, as explained in chapter 7.3 Transition Templates – Making problems absorb into platform.

The contribution of this approach is that the level of detail in the requirements model (problem architecture) and in the corresponding model of architecture design (solution architecture) remains the same and can be maintained.

This is useful for controlling the scope of the requirements, for the dependency management of requirements and architecture as well as for the decision making about solution alternatives and the reuse for functional user requirements.

E.1. Client–Dispatcher–Server

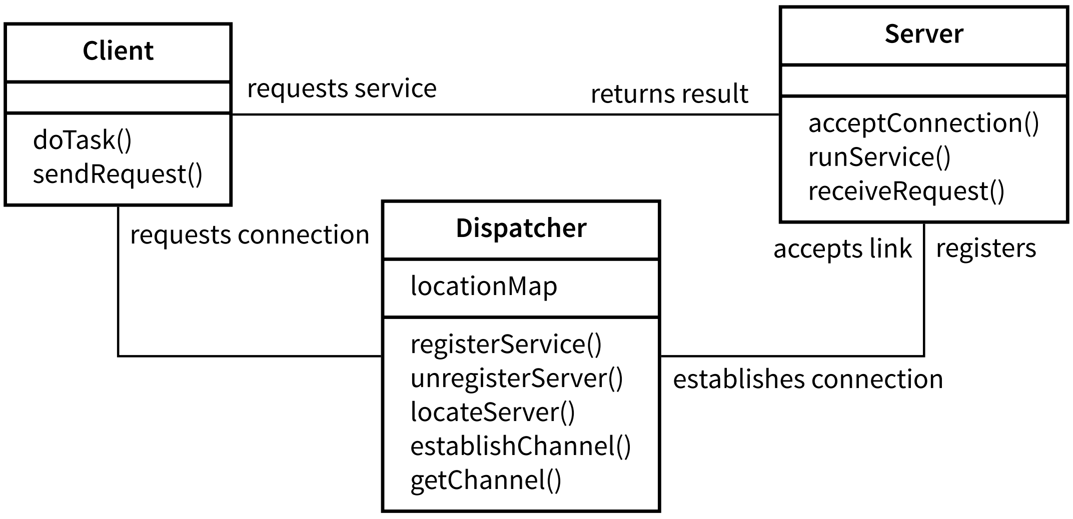


FIGURE E.1 Client–Dispatcher–Server as UML class diagram, adapted from [46, page 326]

E.2. Forwarder–Receiver

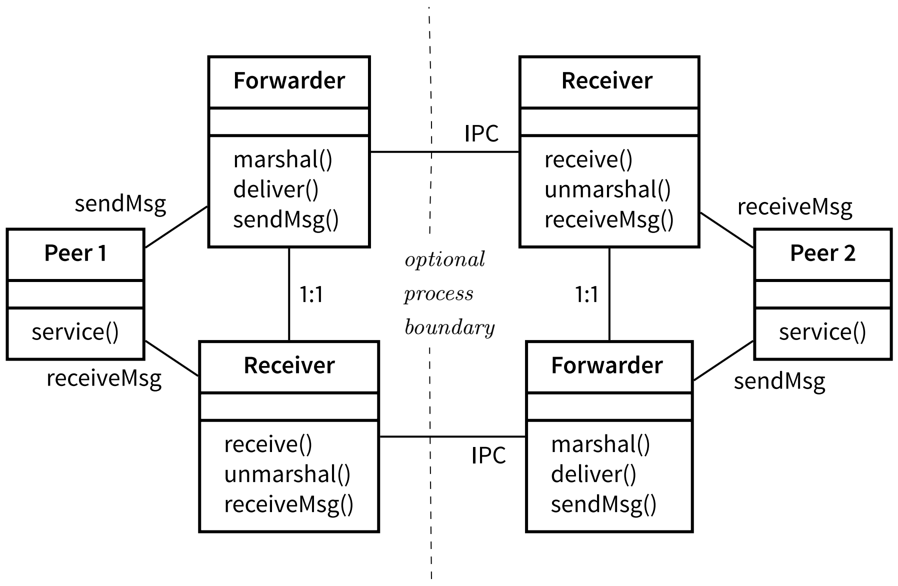


FIGURE E.2 Forwarder–Receiver as UML class diagram, adapted from [46, page 311]

E.3. Observer/Publisher-Subscriber

Note: Neither Gamma et al. in [90] nor Buschmann et al. in [46, page 339ff] specify a UML class diagram-like structure for the static relationships between the participants in a publisher–subscriber design pattern. Both refer to the observer design pattern, when explaining the kind of interaction, which is known as publish-subscribe [90, page 294].

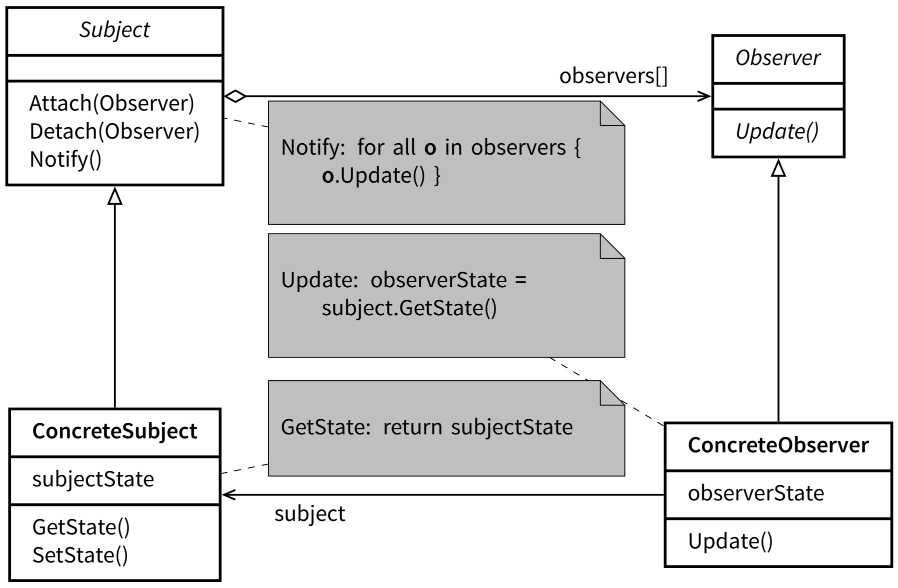


FIGURE E.3 Observer as UML class diagram, adapted from [90, page 294]

E.4. Model–View–Controller

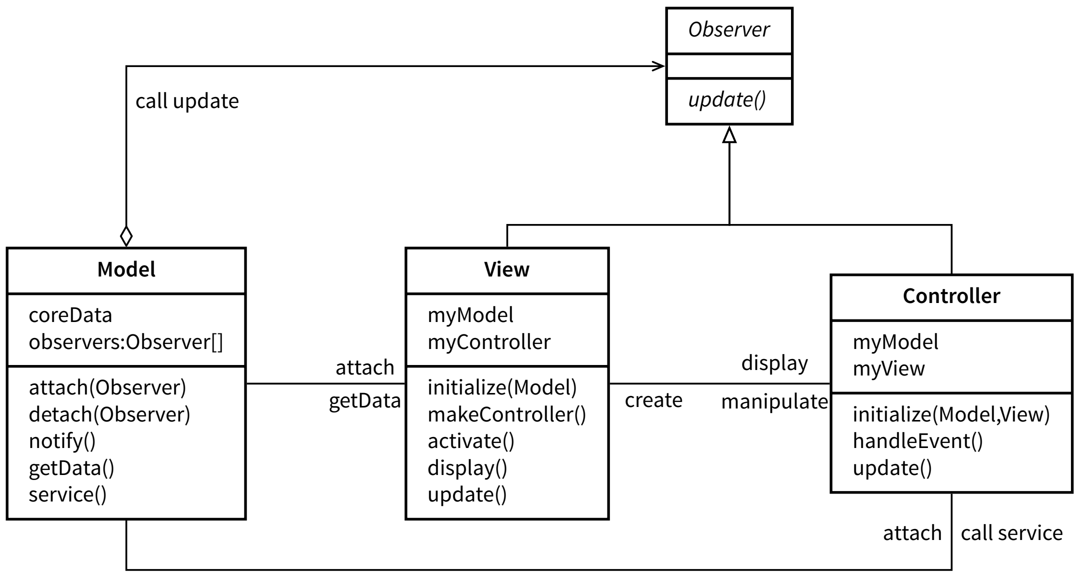


FIGURE E.4 Model–View–Controller as UML class diagram, adapted from [46, page 129]

F. For Further Discussion

This appendix presents some first extensions on the concepts developed and introduced in this dissertation. It is structured as follows: Section F.1 User Story templates out of problem-based functional size measurement patterns on page 291 adapt the SOPHIST's classification of requirements by keywords to problem-based functional size measurement pattern for classifying these by indicating their transactional function. This results problem-based user story templates. Section F.2 Data modeling by problem-based user story templates gives an idea of how to structures data due to use of problem-based user story templates into a UML class diagram. Section F.3 Roles and Permissions matrix by problem-based user story templates discusses permission matrix, which are of use for managing data access rights and for implementing data protection requirements. Section F.4 Story Mapping by problem-based user story templates – The amigos' big picture presents an alternative requirements dependency management by state transition diagrams using an agile practices, namely Storyboards.

F.1. User Story templates out of problem-based functional size measurement patterns

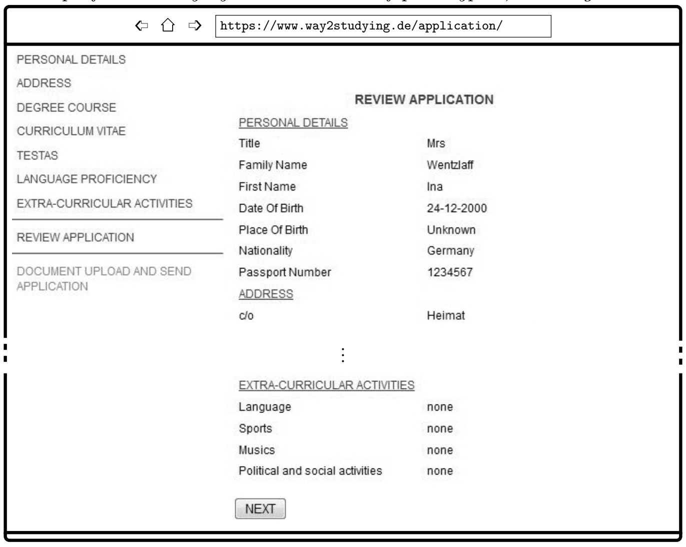
no.	user story template
TFK1.	TF-keyword = create *As a user/in the role of an <role-name="editor">,* *I want to <TF-keyword="create"> <DF="candidate data">,* *so that I can write and read all <DET of DF="details"> on the <DF> later.*
TFK2.	TF-keyword = update *As a user/in the role of an <role-name="editor">,* *I want to <TF-keyword="update"> <DF="candidate data">,* *so that I can keep all <DET of DF="details"> on the <DF> uptodate.*
TFK3.	TF-keyword = delete *As a user/in the role of an <role-name="editor">,* *I want to <TF-keyword="delete"> <DF="candidate data">,* *so that I cannot write or read this <DF> anymore.*
TFK4.	TF-keyword = show *As a user/in the role of an <role-name="editor"> or <role-name="consultant">,* *I want the software application to <TF-keyword="show"> all <DET of DF="details">* *of <DF="candidate data"> (on the display) to me,* *so that I can (only) read all <DET of DF="details"> available on the <DF>.*
TFK5.	TF-keyword = browse *As a user/in the role of an <role-name="editor"> or <role-name="consultant">,* *I want to <TF-keyword="browse"> all available <DF="candidate data">* *that can be limited by <search_ criteria>,* *so that I can choose one <DF> from the search_ results (list)* *to write or read it(s <DET of DF="details">).*
TFK6.	TF-keyword = export *As a user/in the role of an <role-name="editor"> or <role-name="consultant">,* *I want the software application to <TF-keyword="export"> <DF="candidate data">* *via interface/in the format of . . . (<EIF>="Email/Excel-Export"),* *so that . . . (continue working on <DF> in <EIF>) becomes possible (to me).*

TABLE F.1 Problem-based user story templates applying TF-keywords

no.	TF-keyword	(also) write	(only) read	EP	exemplary comparison to OCL expression
TFK1.	create	YES	NO	EI	***collection***→*includes(element)=true*
TFK2.	update	YES	NO	EI	*collection→select(**element**) <u>and</u> element = update(**element**@pre)*
TFK3.	delete	YES	NO	EI	***collection***→*excludes(element)=true*
TFK4.	show	NO	YES	EQ	*collection→select(**element**) <u>or</u> self^display(**element**)*
TFK5.	browse	NO	YES	EQ	***collection***→*collect(element=search_ criteria)=collection(search_ results)*
TFK6.	export	NO	YES	EO	*collection→select(**element**) <u>and</u> other^export(**element|collection**)*

Note:
Transactional **F**unction-keywords serve the same purpose as *signal words* or respective *process verbs* in *requirements templates* proposed by The SOPHISTs [207].

TABLE F.2 List of TF-keywords for problem-based user story templates

structure	**content**	*about*
name	FUR#03: show (candidate) data	*<rwp-no:, TF-keyword, data function>*
keywords	candidate	*this user story is <accessable by role>*
user interface	*mockup of user story by a <screenshot of prototype>, or design sketch* 	

https://www.way2studying.de/application/

PERSONAL DETAILS
ADDRESS
DEGREE COURSE
CURRICULUM VITAE
TESTAS
LANGUAGE PROFICIENCY
EXTRA-CURRICULAR ACTIVITIES
REVIEW APPLICATION
DOCUMENT UPLOAD AND SEND APPLICATION

REVIEW APPLICATION

PERSONAL DETAILS

Title	Mrs
Family Name	Wentzlaff
First Name	Ina
Date Of Birth	24-12-2000
Place Of Birth	Unknown
Nationality	Germany
Passport Number	1234567

ADDRESS

c/o	Heimat

⋮

EXTRA-CURRICULAR ACTIVITIES

Language	none
Sports	none
Musics	none
Political and social activities	none

user story	*created by user story template in accordance with table F.1* *As a user/in the role of an <role-name="**candidate**">,* *I want the <software application="**Student Recruitment Web Portal**">* *to <TF-keyword="**show**"> all <DET of DF="**information**">* *of <DF="**candidate data**"> (on the display) to me,* *so that I can read all <DET of DF="**information**"> available on the <DF="**candidate data**">.*
user acceptance criteria	*acceptance criteria (AC) account for data model, such as discussed in section 13.2.4, workflow (WF)/flow of usage in story map, such as table F.7, and test cases, maybe referenced by test tool*

data model[1]	information (=DET) of "(candidate) data" = (DF): title::={Mr|Mrs}, *family name, *first name, …
WF pre[2]	User story "FUR#02 record candidate data" is successfully completed.
WF post[3]	none
AC.01.[4]	If candidate selects "review application" in the navigation bar, all stored information of "(candidate) data" according to the data model section are "shown" on the display.
AC.02.[5]	If candidate selects "review application" in the navigation bar, but no information about "(candidate) data" according to the data model section is available, an empty form is "shown" on the display, or "review application" is not accessable via the navigation bar.

[1] The data model can be given as a simple list of attributes, by a UML class diagram, etc.
[2] WF pre means, that the mentioned user story is a prerequisite for the execution of the actual user story.
[3] WF post means, that the mentioned user story can be executed after the actual one is successfully completed.
[4] Acceptance criteria must be testable. This requires an recognisable action and the definition of expected output, when the action is successfully completed.
[5] Plausibility checks and respective error message handling/notification of the user can also belong to acceptance criteria.

TABLE F.3 Requirements Work Package as multi-purpose work item

F.2. Data modeling by problem-based user story templates

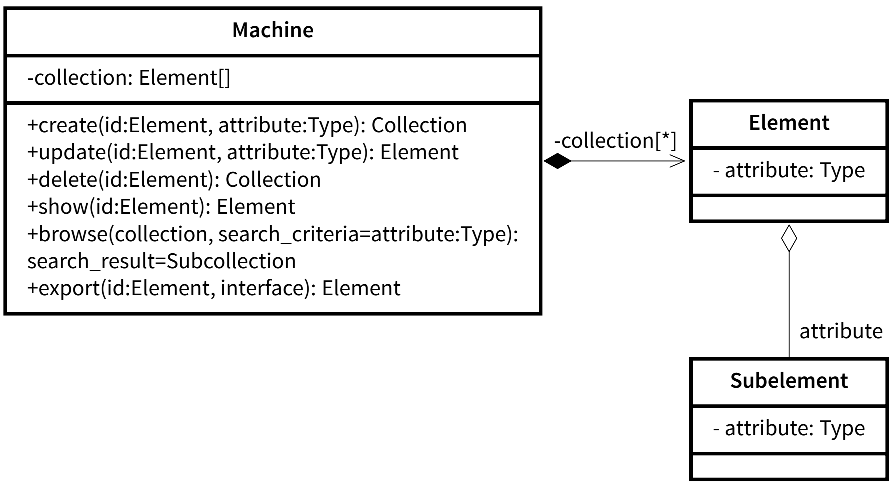

Machine
-collection: Element[]
+create(id:Element, attribute:Type): Collection
+update(id:Element, attribute:Type): Element
+delete(id:Element): Collection
+show(id:Element): Element
+browse(collection, search_criteria=attribute:Type): search_result=Subcollection
+export(id:Element, interface): Element
-collection[*]
Element
- attribute: Type
attribute
Subelement
- attribute: Type

FIGURE F.1 Exemplary Data Model structured according to OCL expressions

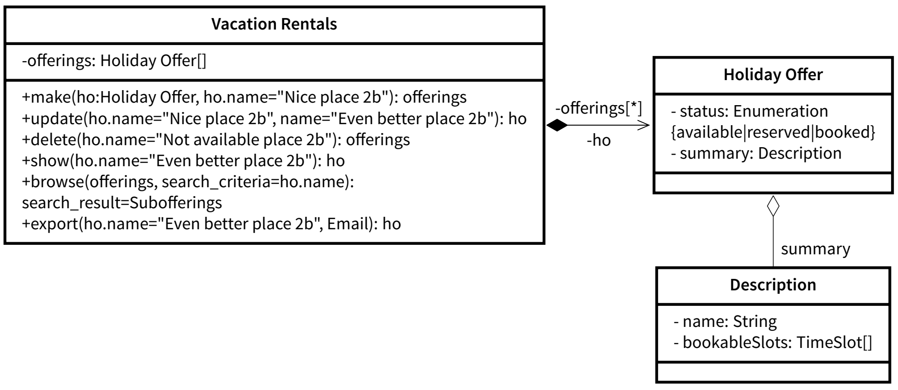

Vacation Rentals
-offerings: Holiday Offer[]
+make(ho:Holiday Offer, ho.name="Nice place 2b"): offerings
+update(ho.name="Nice place 2b", name="Even better place 2b"): ho
+delete(ho.name="Not available place 2b"): offerings
+show(ho.name="Even better place 2b"): ho
+browse(offerings, search_criteria=ho.name): search_result=Subofferings
+export(ho.name="Even better place 2b", Email): ho
-offerings[*]
-ho
Holiday Offer
- status: Enumeration {available|reserved|booked}
- summary: Description
summary
Description
- name: String
- bookableSlots: TimeSlot[]

FIGURE F.2 Exemplary Data Model for the Vacation Rentals

F.3. Roles and Permissions matrix by problem-based user story templates

A user story, which is created by a problem-based user story template, describes functional user requirements, which consist of some functionality and their (involved read and write access on) data. In addition, it must be specified, which user (which role) can access which story, in order to clarify their access rights. This mapping is represented by a permission matrix, see also [29, chapter 11].

Table F.4 outlines the structure of a permission matrix for problem-based user story templates. It maps some general roles to each problem-based user story template, each which is identifiable by a TF-keyword as proposed in section F.1.

For instance, in the context of the Student Recruitment Web Portal the editor is a role which owns full write and read access on the candidate date, while the consultant role is only allowed to access functionality that provides read only access to the candidate data. In this example, the administrator role has full access to functionality, that deals with the management of data about authorized actors.

role/story by TF-keyword	TFK1. create	TFK2. update	TFK3. delete	TFK4. show	TFK5. browse	TFK6. export	data function
editor	YES	YES	YES	YES	YES	YES	$<DF_x = "candidate\ data">$
consultant	NO	NO	NO	YES	YES	YES	
administrator	YES	YES	YES	YES	YES	YES	$<DF_y = "authorized\ actors">$

Legend:
TF = transactional function, which represents the functionality of a user story
DF = data function, which represents the data of a user story

TABLE F.4 Permissions matrix, exemplary structure

Table F.5 takes the UML use case diagram of the Vacation Rentals Web Application into account.

role/story	R01: make **offer** (EI)	R02: browse (offer) EQ	R03 finish book (offer) EI	R04: book (offer) EO	R05: quit book (offer) EI	R06: record pay. (offer) EI	R78: rate (offer) EO	R09: browse (offer) EQ	<DF>
staff	Y	Y	**Y**	Y	Y	N	N	N	offer
guest	N	N	N	N	N	Y	Y	**Y**	

Legend:

Y = role has the right to execute user stories and access respective <DF>
N = role is not allowed to execute user stories and access to respective <DF> via this story is denied
story = write access to data function (DF), otherwise read
columncolor = these stories are the same user story used by different roles only, synergize possible.

TABLE F.5 Permissions matrix for the Vacation Rentals Web Application

Table F.6 takes the UML use case diagram of the Student Recruitment Web Portal into account.

role/story	FUR01 (none|url) EO	FUR02: record (data) EI	FUR03 (data) EQ	FUR04 (data) EO	FUR05:upload (files) EI	FUR06 (resume=filesNdata) EQ
candidate	Y	Y	Y	Y	Y	**N**
(program) admin	**N**	N	N	N	N	**Y**

Legend: see table F.5

TABLE F.6 Permissions matrix for the Student Recruitment Web Portal

F.4. Story Mapping by problem-based user story templates – The amigos' big picture

Table F.7 presents an alternative workflow model as in the state-transition diagram give by figure 8.4 on page 167. In an agile project setting, it can be used as a storyboard that also includes the permission matrix.

no.	role		scenario		
	candidate	admin			
1.	Y	N	FUR01: access application (EO)		
2.	Y	N		FUR02: record **data** (EI)	
3.	Y	N			FUR03: review **data** (EQ) FUR04: download **data** (EO)
4.	Y	N		FUR05: upload **files** (EI)	
5.	N	Y			FUR06: compile resume=**files&data** (EQ)

Legend:
Y = role has the right to execute user stories of this scenario
N = role is not allowed to execute user stories of this scenario
story = write access to data function **files** and **data**, otherwise read only

TABLE F.7 Story Mapping for the Student Recruitment Web Portal

Table F.8 presents an alternative workflow model as in the state-transition diagram give by figure 10.8 on page 215. In an agile project setting, it can be used as a storyboard that also includes the permission matrix by roles[1].

no.	role		scenario			
	staff	guest				
1.	Y	N	R01: make **offer** (EI)			
2.	**Y**	N		R02: browse **offers** (EQ)		
3.	N	**Y**		R09: browse **offers** (EQ)		
4.	N	Y		R03: finish/reserve book **offer** (EI) R05: quit/reset book **offer** (EI) R04: book **offer** (EO)		
5.	Y	N			R06: record pay. for **offer** (EI)	
6.	Y	N				R78: rate **offer** (EO)

Legend:
Y = role has the right to execute user stories of this scenario
N = role is not allowed to execute user stories of this scenario
story = write access to data function **offer**, otherwise read only
rowcolor = these scenarios make use of the same user story by different roles only, synergize possible.

TABLE F.8 Story Mapping for the Vacation Rentals Web Application

[1] Roles are compared here to actors in a UML use case diagram.

Tables

Figures

Examples

Acronyms

AB	Application Boundary
ABP	Architectural Blueprint
ADP	Architectural Design Pattern
APP	Software Application
BFC	Base Functional Component
DET	Data Element Type
DF	Data Function
EI	External Input
EIF	External Interface File
EO	External Output
EP	Elementary Process
EQ	External Inquiry
FP	Function Point
FPA	Function Point Analysis
FPC	Function Point Count
FSM	Functional Size Measurement
FTR	File Type Referenced
FUR	Functional User Requirements
IFPUG	International Function Point Users Group
ILF	Internal Logical File
LCE	Life-Cycle Expression
PI	Primary Intent
RET	Record Element Type
STD	State Transition Diagram
TF	Transactional Function
USR	User

References

[1] Alain Abran, Denis St-Pierre, Marcela Maya, and Jean-Marc Desharnais. Full Function Points for Embedded and Real-Time Software. Technical Report Oct. 30-31, UKSMA Fall Conference, London, UK, 1998.
Cited on page 108.

[2] Allan J. Albrecht. Measuring Application Development Productivity. In Proceedings of the Joint SHARE/GUIDE IBM Applications Development Symposium, pages 83–92, White Plains, New York, 14 – 17 October 1979. IBM Corporation.
Cited on pages 5, 24, and 42.

[3] Azadeh Alebrahim. Bridging the Gap between Requirements Engineering and Software Architecture – A Problem-Oriented and Quality-Driven Method. Dissertation, University of Duisburg-Essen, Software Engineering (Prof. Dr. Maritta Heisel), 2017.
Cited on pages 149 and 171.

[4] Azadeh Alebrahim, Stephan Faßbender, Maritta Heisel, and Rene Meis. Problem-Based Requirements Interaction Analysis. In Proceedings of the International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ), LNCS 8396, pages 200–215. Springer, 2014.
DOI 10.1007/978-3-319-05843-6_15 .
Cited on page 171.

[5] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. A Pattern Language: Towns, Buildings, Construction. Center for Environmental Structure Berkeley, Calif: Center for Environmental Structure series. Oxford University Press, 1978.
Cited on pages 45 and 114.

[6] Glen B. Alleman. The Seven Principles Of Project Performance Management, Jan. 10, 2010. URL https://herdingcats.typepad.com/my_weblog/2010/01/the-seven-principles-of-project-performance-management.html.
Cited on page 280.

[7] Glen B. Alleman. Performance-Based Project Management: Increasing the Probability of Project Success. AMACOM, 2014.
Cited on pages 26 and 46.

[8] Glen B. Alleman. AMACOM Books: Blog – Glen Alleman on Elements of Project Success, February 25, 2014.
URL https://amacombooks.wordpress.com/2014/02/25/glen-alleman-elements-project-success/.
Cited on pages 3 and 7.

[9] Agile Alliance. Glossary: What is Role-Feature-Reason?, 2018.
URL https://www.agilealliance.org/glossary/role-feature/.
Cited on page 41.

[10] Agile Alliance. Glossary: What are the Three Amigos?, 2019.
URL https://www.agilealliance.org/glossary/three-amigos.
Cited on pages 160 and 161.

[11] Agile Alliance. Glossary: What is Story Mapping?, 2020.
URL https://www.agilealliance.org/glossary/storymap.
Cited on page 254.

[12] Scott W. Ambler. The Object Primer: Agile Model-Driven Development with UML 2.0. Cambridge University Press, New York, NY, USA, 3rd edition, 2004.
Cited on pages 114 and 120.

[13] Scott W. Ambler. Acceleration: An agile productivity metric, June 13, 2016.
URL http://www.disciplinedagiledelivery.com/acceleration/.
Cited on pages 3 and 7.

[14] Scott W. Ambler. Usage scenarios: An agile introduction, 2018.
URL http://www.agilemodeling.com/artifacts/usageScenario.htm.
Cited on page 120.

[15] Scott W. Ambler and Mark Lines. Disciplined Agile Delivery: A Practitioner's Guide to Agile Software Delivery in the Enterprise. IBM Press, 1st edition, 2012.
Cited on page 193.

[16] Marc Andreessen. Why Software is Eating The World, August 20, 2011.
URL https://www.wsj.com/articles/SB10001424053111903480904576512250915629460.
Cited on page 2.

[17] Humphreys & Associates. 7 Principles of Earned Value Management Tier 2 System Implementation Intent Guide, August 12, 2013.
URL https://blog.humphreys-assoc.com/7-principles-of-earned-value-management-tier-2-system/.
Cited on page 280.

[18] Atlassian. Products: Jira and Confluence, 2020.
URL https://www.atlassian.com.
Cited on pages 252 and 253.

[19] AXELOS. Managing Successful Projects with PRINCE2®. tso (The Stationery Office Ltd), Norwich, 2009.
URL http://best-management-practice.com.
Cited on pages 6, 7, 8, 20, 26, and 251.

[20] AXELOS. Service Management – IT Infrastructure Library (ITIL)®. http://www.best-management-practice.com/itil, 2014.
URL http://www.best-management-practice.com/itil.
Cited on page 251.

[21] AXELOS. PRINCE2 Agile®. tso (The Stationery Office Ltd), Norwich, 2015.
URL http://best-management-practice.com.
Cited on pages 7, 14, 26, 39, 164, 165, 174, and 251.

[22] Muhammad Ali Babar and Ian Gorton, editors. Proceedings of the Fourth European Conference on Software Architecture (ECSA 2010), volume 6285 of Lecture Notes in Computer Science, Copenhagen, Denmark, August 23-26, 2010. Springer.
DOI 10.1007/978-3-642-15114-9 .
Cited on pages 2 and 310.

[23] Felix Bachmann, Robert L. Nord, and Ipek Ozkaya. Architectural Tactics to Support Rapid and Agile Stability. CrossTalk – the Journal of Defense Software Engineering, 3(3):20–25, 5/6 2012.
URL http://www.crosstalkonline.org/storage/issue-archives/2012/201205/201205-Bachmann.pdf.
Cited on page 150.

[24] Manish Bahl. Fast, but not furious: Accelerating to win in the fourth industrial revolution, 2017.
URL https://www.cognizant.com/perspectives/fast-but-not-furious-accelerating-to-win-in-the-fourth-industrial-revolution.
Cited on page 3.

[25] Sohaib Shahid Bajwa, Çigdem Gencel, and Pekka Abrahamsson. Software Product Size Measurement Methods: A Systematic Mapping Study. In Joint Conference of the International Workshop on Software Measurement and the International Conference on Software Process and Product Measurement (IWSM/Mensura 2014), pages 176–190, Rotterdam, The Netherlands, October 2014.
DOI 10.1109/IWSM.Mensura.2014.24 .
Cited on page 226.

[26] Arthur Bakker, Phillip Kent, Celia Hoyles, and Richard Noss. Designing for communication at work: A case for technology-enhanced boundary objects. International Journal of Educational Research, 50(1):26–32, January 2011.
DOI 10.1016/j.ijer.2011.04.006 .
Cited on pages 38, 41, and 46.

[27] Balsamiq. Wireframes, 2020.
URL https://balsamiq.com/.
Cited on page 253.

[28] Vitor A. Batista, Daniela C. C. Peixoto, Eduardo P. Borges, Wilson Pádua, Rodolfo F. Resende, and Clarindo Isaías P. S. Pádua. ReMoFP: A Tool for Counting Function Points from UML Requirement Models. Advances in Software Engineering, 2011.
DOI 10.1155/2011/495232 .
Cited on pages 108 and 109.

[29] Joy Beatty and Anthony Chen. Visual Models for Software Requirements. Microsoft Press, 2012.
Cited on page 294.

[30] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Principles Behind The Agile Manifesto, 2001.
URL http://agilemanifesto.org/principles.html.
Cited on pages vi, 15, and 280.

[31] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Manifesto for Agile Software Development, 2001.
URL http://agilemanifesto.org/.
Cited on pages viii, 2, 7, and 15.

[32] Vieri del Bianco and Luigi Lavazza. Applying the COSMIC Functional Size Measurement Method to Problem Frames. In Proceedings of the 14th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS, pages 282–290, Washington, DC, USA, 2009. IEEE Computer Society.
DOI 10.1109/ICECCS.2009.25 .
Cited on page 73.

[33] Dick Billows. Project Trade-Offs – Scope, Time, Cost, Risk, Quality, October 12, 2015.
URL https://4pm.com/2015/10/12/project-trade-offs-4pm-com.
Cited on pages 15 and 16.

[34] Barry Boehm. Software Engineering Economics. Prentice Hall, Englewood Cliffs, NJ, 1981.
Cited on pages 4 and 43.

[35] Barry Boehm. Get Ready for Agile Methods, with Care. IEEE Computer, 35(1):64–69, January 2002.
DOI 10.1109/2.976920 .
Cited on pages viii, 8, 9, 10, and 14.

[36] Barry Boehm and Richard Turner. Balancing Agility and Discipline: A Guide for the Perplexed. Addison-Wesley, Boston, MA, USA, 2003.
Cited on page 10.

[37] Jan Bosch. Architecture in the Age of Compositionality. In Babar and Gorton [22], pages 1–4.
DOI 10.1007/978-3-642-15114-9_1 .
URL https://resources.sei.cmu.edu/asset_files/Presentation/2011_017_001_22595.pdf.
Cited on pages 2, 150, and 151.

[38] Jan Bosch. Achieving Simplicity with the Three-Layer Product Model. Computer, 46(11): 34–39, November 2013.
DOI 10.1109/MC.2013.295 .
Cited on page 114.

[39] Jan Bosch. Speed, Data, and Ecosystems: Excelling in a Software-Driven World. CRC Press, Boca Raton, FL, USA, 2016.
Cited on page 2.

[40] Jan Bosch. Software driven world, 2017.
URL https://janbosch.com/blog/index.php/2017/01/28/9-out-of-10-in-rd-work-on-commodity/.
Cited on page 114.

[41] Jan Bredereke. Configuring members of a family of requirements using features. In Feature Interactions in Telecommunications and Software Systems VIII, ICFI'05, 28-30 June 2005, Leicester, UK, pages 96–113, 2005.
Cited on page 38.

[42] Frederick P. Brooks, Jr. No Silver Bullet Essence and Accidents of Software Engineering. Computer, 20(4):10–19, April 1987.
DOI 10.1109/MC.1987.1663532 .
Cited on pages 11, 13, 14, and 26.

[43] Frederick P. Brooks, Jr.
The Mythical Man-Month: Essays on Software Engineering (Anniversary Edition). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.
Cited on page 14.

[44] Nanette Brown, Robert L. Nord, and Ipek Ozkaya. Enabling Agility through Architecture. CrossTalk – the Journal of Defense Software Engineering, 6(6):12–17, 11/12 2010.
URL http://www.crosstalkonline.org/storage/issue-archives/2010/201011/201011-Brown.pdf.
Cited on page 150.

[45] Frank Buschmann and Kevlin Henney. Software architecture styles and paradigms. SIG DATACOM OOP 2011 – Software meets Business, January 24-28, 2011, Munich, Germany, 2011.
URL http://www.sigs.de/download/oop_2011/downloads/files/Fr2_Buschmann_Henney_Architecture_Styles_And_Paradigms.pdf.
Cited on page 149.

[46] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. Pattern-Oriented Software Architecture – A System of Patterns. John Wiley & Sons, Chichester, USA, 1996.
Cited on pages vii, 137, 139, 141, 143, 145, 282, 287, 288, 289, 298, 300, and 301.

[47] Rina Diane Caballar. Programming Without Code: The Rise of No-Code Software Development, March 11, 2020.
URL https://spectrum.ieee.org/tech-talk/computing/software/programming-without-code-no-code-software-development.
Cited on page vi.

[48] Robert C. Camp. Benchmarking – The Search for Industry Best Practices that Lead to Superior Performance. American Society for Quality Control, 1989.
Cited on pages 26, 175, and 193.

[49] Christine Choppy and Maritta Heisel. Une approache à base de patrons pour la spécification et le développement de systèmes d'information. Approches Formelles dans l'Assistance au Développement de Logiciels - AFADL, 2004.
Cited on pages 64 and 70.

[50] Christine Choppy, Denis Hatebur, and Maritta Heisel. Architectural patterns for problem frames. Software, IEE Proceedings, 152(4):198 – 208, 09 2005.
DOI 10.1049/ip-sen:20045061 .
Cited on page 149.

[51] Christine Choppy, Denis Hatebur, and Maritta Heisel. Composing architectures based on architectural patterns for problem frames. Technical Report , , 2005.
URL https://www.uni-due.de/imperia/md/content/swe/papers/2005comparch.pdf.
Cited on pages 159 and 171.

[52] Christine Choppy, Denis Hatebur, and Maritta Heisel. Component Composition through Architectural Patterns for Problem Frames. In 13th Asia-Pacific Software Engineering Conference (APSEC'06), pages 27–36. IEEE Computer Society, 2006.
Cited on pages 159 and 171.

[53] Jane Cleland-Huang, Brian Berenbach, Stephen Clark, Raffaella Settimi, and Eli Romanova. Best Practices for Automated Traceability. Computer, 40(6):27–35, June 2007.
DOI 10.1109/MC.2007.195 .
Cited on page 151.

[54] Mike Cohn. User Stories Applied: For Agile Software Development. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.
Cited on pages 4, 39, and 43.

[55] Mike Cohn. Agile Estimating and Planning. Prentice Hall Professional Technical Reference, Upper Saddle River, NJ, USA, 2005.
Cited on pages 4, 5, 13, 19, 26, 43, and 299.

[56] Mike Cohn. Establishing a Common Baseline for Story Points, August 6, 2008.
URL http://www.mountaingoatsoftware.com/blog/establishing-a-common-baseline-for-story-points.
Cited on pages 4 and 5.

[57] Mike Cohn. User Stories, Epics and Themes, October 24, 2011.
URL https://www.mountaingoatsoftware.com/blog/stories-epics-and-themes.
Cited on page 39.

[58] Mike Cohn. The Best Way to Establish a Baseline When Playing Planning Poker, May 31, 2016.
URL http://www.mountaingoatsoftware.com/blog/the-best-way-to-establish-a-baseline-when-playing-planning-poker.
Cited on pages 4 and 73.

[59] Mike Cohn. How to Prevent Estimate Inflation, May 3, 2016.
URL http://www.mountaingoatsoftware.com/blog/how-to-prevent-estimate-inflation.
Cited on page 4.

[60] Mike Cohn. User Stories and User Story Examples, 2018.
URL http://www.mountaingoatsoftware.com/agile/user-stories.
Cited on pages 4 and 39.

[61] Derek Coleman, Patrick Arnold, Stephanie Bodoff, Chris Dollin, Helena Gilchrist, Fiona Hayes, and Paul Jeremaes. Object-Oriented Development: The Fusion Method. Prentice Hall, 1994.
Cited on page 158.

[62] Jim Collins. Built to Flip, March 2000.
URL https://www.jimcollins.com/article_topics/articles/built-to-flip.html.
Cited on page 257.

[63] Jim Collins. Good to great : why some companies make the leap ... and others don't. Harper Business, New York, NY, 2001.
Cited on page 257.

[64] James O. Coplien, Daniel Hoffman, and David M. Weiss. Commonality and Variability in Software Engineering. IEEE Software, 15(6):37–45, November 1998.
DOI 10.1109/52.730836 .
Cited on pages 3 and 38.

[65] COSMIC. The COSMIC Implementation Guide for ISO/IEC 19761:2011 Software engineering. A functional size measurement method, April 2015.
URL http://www.cosmicon.com/portal/public/MMv4.0.1.pdf.
Cited on page 45.

[66] COSMIC. ISO/IEC 19761:2011 Software engineering. A functional size measurement method, 2016.
URL http://www.cosmicon.com/.
Cited on page 45.

[67] Isabelle Côté. A Systematic Approach to Software Evolution. Dissertation, University of Duisburg-Essen, Software Engineering (Prof. Dr. Maritta Heisel), 2013.
Cited on pages 30 and 196.

[68] Isabelle Côté, Denis Hatebur, Maritta Heisel, Holger Schmidt, and **Ina Wentzlaff**. A Systematic Account of Problem Frames. In Proceedings of the 12th European Conference on Pattern Languages of Programs (EuroPLoP 2007), pages 749–767, Irsee, Germany, July 4-8, 2007. Universitätsverlag Konstanz.
URL http://www.uni-due.de/imperia/md/content/swe/papers/2007europlop.pdf.
Cited on pages 23, 27, 36, 50, 53, 69, 70, 71, 72, 268, and 297.

[69] Isabelle Côté, Maritta Heisel, and **Ina Wentzlaff**. Pattern-Based Evolution of Software Architectures. In Flávio Oquendo, editor, Proceedings of the First European Conference on Software Architecture (ECSA 2007), volume 4758 of Lecture Notes in Computer Science, pages 29–43, Aranjuez, Spain, September 24-26, 2007. Springer.
DOI 10.1007/978-3-540-75132-8_4 . **Best Paper Award**.
Cited on pages 23, 30, 31, and 149.

[70] Isabelle Côté, Maritta Heisel, and **Ina Wentzlaff**. Pattern-Based Exploration of Design Alternatives for the Evolution of Software Architectures. International Journal of Cooperative Information Systems (IJCIS), 16(3/4):341–365, September/December 2007.
DOI 10.1142/S0218843007001688 .
Cited on pages 23, 31, 149, and 158.

[71] National Research Council. Measuring Performance and Benchmarking Project Management at the Department of Energy. The National Academies Press, Washington, DC, 2005.
DOI 10.17226/11344 .
URL https://www.nap.edu/catalog/11344/measuring-performance-and-benchmarking-project-management-at-the-department-of-energy.
Cited on page 193.

[72] Cucumber. Gherkin, 2020.
URL https://cucumber.io/docs/gherkin/.
Cited on page 253.

[73] Maya Daneva, Egbert van der Veen, Chintan Amrit, Smita Ghaisas, Klaas Sikkel, Ramesh Kumar, Nirav Ajmen, Uday Ramteerthkar, and Roel Wieringa. Agile requirements prioritization in large-scale outsourced system projects: An empirical study. Journal of Systems and Software, 86(5):1333–1353, 2013.
Cited on pages vii, 4, 5, and 73.

[74] Tom DeMarco. Structured Analysis and System Specification. Yourdon Press, 1979.
Cited on page 115.

[75] Tom DeMarco. The deadline: a novel about project management. Dorset House Publishing, 1997.
Cited on page 249.

[76] William Edwards Deming. Out of the Crisis. Massachusetts Institute of Technology, Center for Advanced Engineering Study, 2000.
Cited on pages vii and 176.

[77] Der Beauftragte der Bundesregierung für die Informationstechnik. V-Modell XT Bund, 2020.
URL https://www.cio.bund.de/SharedDocs/Publikationen/DE/Architekturen-und-Standards/V-Modell-XT-Bund/v_modell_xt_bund_dokumentation_23.pdf. (available in german only).
Cited on page 251.

[78] Edsger Wybe Dijkstra. The Humble Programmer. Communications of the ACM, 15(10): 859–866, 1972.
URL https://www.cs.utexas.edu/~EWD/ewd03xx/EWD340.PDF. Turing Award lecture.
Cited on page v.

[79] Edsger Wybe Dijkstra. On the cruelty of really teaching computing science, December 2, 1988.
URL https://www.cs.utexas.edu/~EWD/ewd10xx/EWD1036.PDF. EWD 1036.
Cited on page vi.

[80] George Dinwiddie. If you don't automate acceptance tests, 2009.
URL http://blog.gdinwiddie.com/2009/06/17/if-you-dont-automate-acceptance-tests/.
Cited on pages 160 and 161.

[81] Docker. Containerization, 2020.
URL https://www.docker.com/resources/what-container.
Cited on page 253.

[82] George T. Doran. There's a S.M.A.R.T. way to write management's goals and objectives. Management Review, 70:35–36, 1981.
URL https://community.mis.temple.edu/mis0855002fall2015/files/2015/10/S.M.A.R.T-Way-Management-Review.pdf.
Cited on pages 19, 20, 21, 39, 174, 179, and 297.

[83] Christof Ebert and Reiner Dumke. Software Measurement: Establish - Extract - Evaluate - Execute. Springer, 2007.
DOI 10.1007/978-3-540-71649-5 .
Cited on page 3.

[84] Eric Evans. Domain-Driven Design: Tacking Complexity In the Heart of Software. Addison-Wesley, MA, USA, 2003.
Cited on page 253.

[85] Christoph Fehling, Frank Leymann, Ralph Retter, Walter Schupeck, and Peter Arbitter. Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications. Springer, 2014.
URL http://www.cloudcomputingpatterns.org/.
Cited on pages 284 and 298.

[86] Ryan Fogarty. Postmodernism in Software Development, 2009.
URL https://well.tc/3e39.
Cited on pages vi, 191, and 246.

[87] Martin Fowler. ProjectionalEditing, January 14, 2008.
URL https://martinfowler.com/bliki/ProjectionalEditing.html.
Cited on page 252.

[88] Jakob Freund and Bernd Rücker. Camunda – Workflow and Decision Automation, 2021.
URL https://camunda.com/.
Cited on page 254.

[89] Andrei Furda, Colin Fidge, Alistair Barros, and Olaf Zimmermann. Reengineering Data-Centric Information Systems for the Cloud – A Method and Architectural Patterns Promoting Multitenancy, chapter 13, pages 227–251. Volume 1 of Mistrik et al. [157], 2017.
DOI 10.1016/B978-0-12-805467-3.00013-2 .
URL https://www.sciencedirect.com/science/article/pii/B9780128054673000132.
Cited on page 145.

[90] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns – Elements of Reusable Object-Oriented Software. Addison Wesley, Boston, USA, 1995.
Cited on pages vii, 45, 114, 137, 143, 283, 288, 298, and 301.

[91] David Garlan and Mary Shaw. An Introduction to Software Architecture. Technical Report CMU/SEI-94-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1994.
URL http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=12235.
Cited on pages vii and 155.

[92] David Garmus and David Herron. Function Point Analysis - Measurement Practices for Successful Software Projects. Addison-Wesley, 2001.
Cited on pages 24 and 42.

[93] Kai T. Gilb. Concept Glossary for projects, planning, management, development, delivery and maintenance, and for specification-quality-control, 2013.
URL http://concepts.gilb.com/Glossary.
Cited on page 20.

[94] Tom Gilb. Competitive Engineering: A Handbook For Systems Engineering, Requirements Engineering, and Software Engineering Using Planguage. Elsevier Science, 2005.
Cited on pages 21 and 22.

[95] Tom Gilb. Value Planning. Leanpub, 2015.
URL tinyurl.com/ValuePlanning.
Cited on page 19.

[96] Kesten C. Green and J. Scott Armstrong. Structured analogies for forecasting. International Journal of Forecasting, 23(3):365 – 376, 2007.
DOI https://doi.org/10.1016/j.ijforecast.2007.05.005 .
Cited on page 251.

[97] James Grenning. Planning Poker or How to avoid analysis paralysis while release planning, 2002.
URL https://wingman-sw.com/papers/PlanningPoker-v1.1.pdf.
Cited on page 43.

[98] Carl A. Gunter, Elsa L. Gunter, Michael Jackson, and Pamela Zave. A reference model for requirements and specifications. IEEE Software, 17(3):37–43, May 2000.
DOI 10.1109/52.896248 .
Cited on page 38.

[99] Jon G. Hall and Lucia Rapanotti. Problem Frames for Socio-technical Systems. Technical Report No: 2003/09, Department of Computing, The Open University, 2003.
Cited on page 149.

[100] Jon G. Hall, Lucia Rapanotti, and Michael Jackson. Problem-Oriented Software Engineering. Technical Report No: 2010/03, Department of Computing, The Open University, 2010.
URL http://computing-reports.open.ac.uk/2010/TR2010-03.pdf.
Cited on pages 37, 38, and 42.

[101] Denis Hatebur. Pattern and Component-based Development of Dependable Systems. Dissertation, University of Duisburg-Essen, Software Engineering (Prof. Dr. Maritta Heisel), 2012.
Cited on page 158.

[102] Denis Hatebur, Maritta Heisel, and Holger Schmidt. Security engineering using problem frames. In Günter Müller, editor, Emerging Trends in Information and Communication Security, pages 238–253, Berlin, Heidelberg, 2006. Springer.
DOI 10.1007/11766155_17 .
Cited on page 29.

[103] Will Hayes. Agile Metrics: Seven Categories (SEI Blog), September 22, 2014.
URL https://insight.sei.cmu.edu/sei_blog/2014/09/agile-metrics-seven-categories.html.
Cited on page 7.

[104] Claudia Hazan. The 13 Mistakes of Function Point Counting, chapter 11, pages 197–214. Volume 2012 of IFPUG [120], 2012.
Cited on pages 74 and 109.

[105] Vikas Hazrati. Is Measuring Hyper-Productivity a Waste of Time?, June 2, 2009.
URL https://www.infoq.com/news/2009/06/measuring-hyper-productivity.
Cited on pages 2 and 3.

[106] Maritta Heisel. Agendas – a concept to guide software development activities. In R. N. Horspool, editor, Systems Implementation 2000: IFIP TC2 WG2.4 Working Conference on Systems Implementation 2000: Languages, methods and tools 23–26 February 1998, Berlin, Germany, pages 19–32, Boston, MA, 1998. Springer US.
DOI 10.1007/978-0-387-35350-0_2 .
Cited on pages 81 and 174.

[107] Maritta Heisel. UML4PF – eclipse plugin, 2017.
URL http://www.uml4pf.org/.
Cited on pages 47, 87, 108, 264, and 278.

[108] Maritta Heisel. Softwaretechnik – lecture notes, 2017.
URL http://swe.uni-due.de/.
Cited on pages 158, 196, 199, 200, 201, and 213.

[109] Maritta Heisel and Michael Goedicke. GenEDA – Generation and Evaluation of Design Alternatives for Software Architectures, 2017.
URL http://www.geneda.org/. GenEDA is funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under the grants no. GO774/5-1, GO774/5-2, HE3322/4-1, and HE3322/4-2.
Cited on page 174.

[110] Maritta Heisel and Denis Hatebur. A model-based development process for embedded systems. In Proceedings of the Workshop on Model-Based Development of Embedded Systems, number TUBS-SSE-2005-01 in . Technical University of Braunschweig, 2005.
URL https://www.uni-due.de/imperia/md/content/swe/papers/2005mbees.pdf.
Cited on page 158.

[111] James A. Highsmith. Agile Software Development Ecosystems, volume 13 of Agile software development series. Addison-Wesley Professional, 2002.
Cited on pages 9 and 18.

[112] Gregor Hohpe and Bobby Woolf. Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley Professional, Boston, MA, USA, 2003.
URL http://www.enterpriseintegrationpatterns.com/patterns/messaging/toc.html.
Cited on pages 286 and 298.

[113] Jacky A. Holloway, C. Matthew Hinton, David Mayle, and Graham Francis. Why benchmark? understanding the processes of best practice benchmarking, 1997.
URL https://www.researchgate.net/profile/C_Hinton/publication/254519995_WHY_BENCHMARK_UNDERSTANDING_THE_PROCESSES_OF_BEST_PRACTICE_BENCHMARKING/links/5755c66008aec74acf5833ba/WHY-BENCHMARK-UNDERSTANDING-THE-PROCESSES-OF-BEST-PRACTICE-BENCHMARKING.pdf.
Cited on pages 175, 193, and 249.

[114] Watts S. Humphrey. A Discipline for Software Engineering. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.
Cited on pages 3, 4, 5, 24, 42, 76, 77, 109, 164, and 297.

[115] Watts S. Humphrey. Using a defined and measured personal software process. IEEE Software, 13(3):77–88, May 1996.
DOI 10.1109/52.493023 .
Cited on pages 76 and 77.

[116] Axel Hunger. Student Recruitment Web Portal, 2020.
URL http://www.way2studying.de/application/.
Cited on page 217.

[117] IFPUG. Software and systems engineering - Software measurement – IFPUG functional size measurement method 2009. International Organization for Standardization, Geneva, Switzerland, 2009.
Cited on pages 5, 24, 28, 37, 43, 44, 46, 51, 55, 57, 64, 65, 76, 78, 80, 81, 82, 83, 93, 94, 109, 226, 262, 263, 264, 265, 266, 267, 268, 269, 270, 271, 272, 273, 274, 275, 276, 277, 278, 298, and 299.

[118] IFPUG. IFPUG CPM 4.3.1 – Function Point Counting Practices Manual (CPM) Version 4.3.1. International Function Point Users Group, Princeton Junction/NJ, USA, 2010.
Cited on pages 5, 44, 51, 57, and 269.

[119] IFPUG. Software Non-functional Assessment Practices (SNAP) Manual 2.1. International Function Point Users Group, Princeton Junction/NJ, USA, 2012.
Cited on page 43.

[120] IFPUG. The IFPUG Guide to IT and Software Measurement, volume 2012. Taylor & Francis Group, 2012.
Cited on pages 109, 317, and 321.

[121] IFPUG. Criteria for Certification of Function Point Software Type 2 (Word) – Expert System That Aids Counting of Function Points, 2017.
URL http://www.ifpug.org/certification/software-certification/.
Cited on pages 108, 264, and 278.

[122] Masaaki Imai. Kaizen: The Key To Japan's Competitive Success. McGraw-Hill Education, 1986.
Cited on page 175.

[123] CMMI Institute. Capability Maturity Model Integration (CMMI)®, 2016.
URL http://cmmiinstitute.com.
Cited on page 279.

[124] Project Management Institute.
A Guide to the Project Management Body of Knowledge (PMBOK Guide). PMI, 2013.
Cited on pages 7, 8, 11, 26, and 251.

[125] Michael A. Jackson. System Development. Prentice-Hall, 1983.
URL `http://mcs.open.ac.uk/mj665/JSPDDevt.pdf`.
Cited on page vii.

[126] Michael A. Jackson. Software Requirements & Specifications: a lexicon of practice, principles and prejudices. Addison-Wesley, New York, NY, USA, 1995.
Cited on pages viii and 40.

[127] Michael A. Jackson. Problem Analysis Using Small Problem Frames. In Special Issue on the 4th Workshop on Formal and Applied Computer Science (WOFACS 1998), volume 22, pages 47–60. South African Computer Journal, 1999.
Cited on page 46.

[128] Michael A. Jackson. Problem Frames: Analysing and Structuring Software Development Problems. Addison-Wesley Professionals, 2001.
Cited on pages viii, 24, 27, 37, 38, 40, 41, 50, 55, 57, 58, 61, 70, 75, 76, 87, 103, 104, 106, 120, 121, 123, 134, 268, and 278.

[129] Ron Jeffries. Essential XP: Card, Conversation, Confirmation, August 30, 2001.
URL `http://xprogramming.com/articles/expcardconversationconfirmation/`.
Cited on page 39.

[130] Ralph E. Johnson. Frameworks = Components + Patterns – How frameworks compare to other object-oriented reuse techniques. Communications of the ACM, 40(10):39–42, Oct. 1997.
DOI `10.1145/262793.262799` .
Cited on page 114.

[131] Capers Jones. The Mess of Software Metrics. Namcook Analytics, Inc., May 2017.
URL `http://www.namcook.com`.
Cited on page 42.

[132] Brett King. Bank 2.0: How Customer Behaviour and Technology Will Change the Future of Financial Services. Marshall Cavendish Business, 2010.
Cited on page 2.

[133] Philippe Kruchten. Architectural blueprints – the 4+1 view model of software architecture. IEEE Software, 12(6):42–50, 1995.
DOI `10.1109/52.469759` .
URL `http://www.cs.ubc.ca/~gregor/teaching/papers/4+1view-architecture.pdf`.
Cited on pages 25, 152, 154, 155, 160, 161, 162, 163, 164, 297, and 300.

[134] Philippe Kruchten. Software Design in a Postmodern Era. IEEE Software, pages 16–18, 2005.
Cited on pages v, vi, and vii.

[135] Philip A. Laplante. What Every Engineer Should Know about Software Engineering. CRC Press, 2007.
Cited on page 151.

[136] Luiz A. Laranjeira. Software Size Estimation of Object-Oriented Systems. IEEE Transactions of Software Engineering, 16(5):510–522, May 1990.
DOI 10.1109/32.52774 .
Cited on page 13.

[137] Grant Larsen. Designing Component-Based Frameworks Using Patterns in the UML. Communications of the ACM, 42(10):38–45, 10 1999.
DOI 10.1145/317665.317674 .
Cited on page 114.

[138] Soren Lauesen. Software Requirements: Styles and Techniques. Pearson Education, 2002.
Cited on page 38.

[139] Soren Lauesen. Software requirements, 2018.
URL http://www.itu.dk/~slauesen/SorenReqs.html.
Cited on page 38.

[140] Luigi Lavazza. Business goals, user needs, and requirements: A problem frame-based view. Expert Systems, 30(3):215–232, 2013.
DOI 10.1111/j.1468-0394.2012.00648.x .
Cited on pages 38 and 41.

[141] Luigi Lavazza and Vieri del Bianco. Functional size measurement based on problem frames: A case study. In Proceedings of the 3rd International Workshop on Applications and Advances of Problem Frames, IWAAPF, pages 44–47, New York, NY, USA, 2008. ACM.
DOI 10.1145/1370811.1370820 .
Cited on page 73.

[142] Luigi Lavazza and Sandro Morasca. Measuring the Functional Size of Real-Time and Embedded Software: a Comparison of Function Point Analysis and COSMIC. In Proceedings of the 8th International Conference on Software Engineering Advances, ICSEA, 2013.
Cited on page 108.

[143] Richard Lawrence. How to split a user story, October 28, 2009.
URL https://agileforall.com/resources/how-to-split-a-user-story/.
Cited on page 251.

[144] Dean Leffingwell. Scaling Software Agility: Best Practices for Large Enterprises. Addison-Wesley Professional, 2007.
Cited on pages 8, 15, 150, and 299.

[145] Susan Leigh Star and James R. Griesemer. Institutional ecology, ‘translations’ and boundary objects: Amateurs and professionals in berkeley’s museum of vertebrate zoology, 1907-39. Social Studies of Science, 19:387–420, August 1989.
DOI 10.1177/030631289019003001 .
Cited on pages 41 and 46.

[146] Mark Leveson. Stable Teams Really Do Matter. Agile Pain Relief Consulting, 2013.
URL http://agilepainrelief.com/notesfromatooluser/2013/09/stable-teams-really-do-matter.html.
Cited on pages 14 and 43.

[147] Stefan Lindegaard. First Mover or Fast Follower: A Key Question for Innovation, Nov. 1, 2014. URL https://www.linkedin.com/pulse/20141101124643-46249-first-mover-or-fast-follower-a-key-question-for-innovation.
Cited on page 2.

[148] The Clever Product Manager. The Three Forms of Waste – Muda, Mura, and Muri, May 19, 2015. URL http://www.cleverpm.com/2015/05/19/the-three-forms-of-waste-muda-mura-and-muri/.
Cited on page 279.

[149] Antony Marcano. How the industry broke the Connextra Template, August 31, 2016. URL http://antonymarcano.com/blog/2016/08/how-the-industry-broke-the-connextra-template/.
Cited on page 41.

[150] James Martin. Application Development without Programmers. Prentice Hall PTR, 1982.
Cited on page vi.

[151] Steve McConnell. Rapid Development: Taming Wild Software Schedules. Microsoft Press, Redmond, WA, USA, 1996.
Cited on pages 9, 10, 11, 12, 15, 16, 18, 21, 22, 26, 250, and 299.

[152] Steve McConnell. Software Estimation: Demystifying the Black Art. Microsoft Press, Redmond, WA, USA, 2006.
Cited on pages 12, 13, 76, and 299.

[153] Rene Meis. Problem-based Privacy Analysis (ProPAn) – A Computer-aided Privacy Requirements Engineering Method. Dissertation, University of Duisburg-Essen, Software Engineering (Prof. Dr. Maritta Heisel), Dec 2018.
URL https://duepublico2.uni-due.de/receive/duepublico_mods_00047797.
Cited on page 257.

[154] Roberto Meli. Software Measurement in Procurement Contracts, chapter 29, pages 561–583. Volume 2012 of IFPUG [120], 2012.
Cited on pages 42 and 74.

[155] Total Metrics. Function Point Frequently Asked Questions, 2017.
URL http://www.totalmetrics.com/function-point-resources/function-point-FAQ.
Cited on page 109.

[156] Bertrand Meyer. Agile!: The Good, the Hype and the Ugly. Springer, 2014.
Cited on pages 3, 114, 149, 174, 176, and 279.

[157] Ivan Mistrik, Rami Bahsoon, Nour Ali, Maritta Heisel, and Bruce Maxim, editors. Software Architecture for Big Data and the Cloud, volume 1. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2017.
URL https://www.sciencedirect.com/science/book/9780128054673.
Cited on pages 145 and 315.

[158] Ben Morris. Agile velocity is not a measure of productivity, August 12, 2013.
URL http://www.ben-morris.com/agile-velocity-is-not-a-measure-of-productivity/.
Cited on page 3.

[159] James Noble and Robert Biddle. Notes on Postmodern Programming. SIGPLAN Notices, 39 (12):40–56, December 2004.
DOI 10.1145/1052883.1052890 .
Cited on pages vi, vii, and 151.

[160] Robert L. Nord, Ipek Ozkaya, Nanette Brown, and Raghvinder S. Sangwan. Modeling Architectural Dependencies to Support Software Release Planning. In S. D. Eppinger, M. Maurer, K. Eben, and U. Lindemann, editors, 13th International Dependency and Structure Modelling Conference, DSM'11, Section: Software Architectures, pages 159–171, Cambridge, Massachusetts, USA, September 14–15, 2011. Design Society.
URL https://www.designsociety.org/download-publication/30831/Modeling+Architectural+Dependencies+to+Support+Software+Release+Planning.
Cited on page 156.

[161] David Norfolk. Continuous Engineering, 8th March, 2014.
URL http://www.bloorresearch.com/blog/the-norfolk-punt/continuous-engineering/.
Cited on page 193.

[162] Bashar Nuseibeh. Weaving Together Requirements and Architectures. IEEE Computer, 34(3): 115–117, March 2001.
DOI 10.1109/2.910904 .
Cited on pages 114 and 117.

[163] International Institute of Business Analysis and Agile Alliance. Agile Extension to the *BABOK*™ Guide, 2017.
URL https://www.agilealliance.org/wp-content/uploads/2017/08/AgileExtension_V2-Member-Copy.pdf.
Cited on pages 253 and 254.

[164] Neil C. Olsen. Survival of the Fastest: Improving Service Velocity. IEEE Software, 12(5):28–38, September 1995.
DOI 10.1109/52.406754 .
Cited on pages 2, 5, 10, and 18.

[165] OMG. Business Process Model And Notation™ Specification, December 9, 2013.
URL https://www.omg.org/spec/BPMN/.
Cited on pages 159 and 254.

[166] OMG. Automated Function Points, Version 1.0, January 2014.
URL http://www.omg.org/spec/AFP/.
Cited on page 42.

[167] OMG. Object Constraint Language® Specification, February 3, 2014.
URL https://www.omg.org/spec/OCL/.
Cited on page 256.

[168] OMG. Unified Modeling Language® Specification, December 5, 2017.
URL https://www.omg.org/spec/UML/.
Cited on pages 39, 122, 123, 168, 169, 208, and 256.

[169] Cristina Palomares, Carme Quer, and Xavier Franch. Requirements reuse and requirement patterns: a state of the practice survey. Empirical Software Engineering, pages 1–44, 2016.
DOI 10.1007/s10664-016-9485-x .
Cited on pages 73 and 74.

[170] European Parliament and Council of the European Union. Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation), May 25, 2018.
URL https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX%3A32016R0679.
Cited on page 257.

[171] David L. Parnas. On the criteria to be used in decomposing systems into modules. Commun. ACM, 15(12):1053–1058, December 1972.
DOI 10.1145/361598.361623 .
Cited on page 38.

[172] David L. Parnas. On the design and development of program families. IEEE Transactions of Software Engineering, 2(1):1–9, 1976.
DOI 10.1109/TSE.1976.233797 .
Cited on page 38.

[173] David L. Parnas. Stop the Numbers Game. Communications of the ACM, 50(11):19–21, November 2007.
DOI 10.1145/1297797.1297815 .
Cited on page 27.

[174] Tom Peters. Thriving on Chaos: Handbook for a Management Revolution. Collins, 1987.
Cited on page 175.

[175] Shari L. Pfleeger, Felicia Wu, Rosalind Lewis, and United States. Air Force. Software Cost Estimation and Sizing Methods: Issues, and Guidelines. Rand Corporation, 2005.
Cited on pages 16, 42, and 299.

[176] John Pocknell. Fast and furious agile development, September 15, 2016.
URL https://www.quest.com/community/b/en/posts/fast-and-furious-agile-development.
Cited on page 3.

[177] Marsha Pomeroy-Huff, Robert Cannon, Timothy A. Chick, Julia Mullaney, and William Nichols. The personal software process^{sm} body of knowledge, version 2.0, August 2009.
URL https://resources.sei.cmu.edu/asset_files/SpecialReport/2009_003_001_15029.pdf.
Cited on pages 76, 77, and 297.

[178] Lawrence H. Putnam and Ware Myers. Measures for Excellence: Reliable Software on Time, within Budget. Yourdon Press, 1992.
Cited on pages 3 and 193.

[179] Rally Software Development. The Impact of Agile – Quantified, 2013.
URL http://www.rallydev.com/finally-get-real-data-about-benefits-adopting-agile.
Cited on page 14.

[180] Rally Software Development. Release Planning Guide, 2013.
URL https://www.projectmanagement.com/pdf/ReleasePlanningGuide.pdf.
Cited on pages 178 and 300.

[181] Brian Randell, Peter Naur, and John Noel Buxton. The NATO Software Engineering Conferences, 1968 and 1969.
URL http://homepages.cs.ncl.ac.uk/brian.randell/NATO/index.html.
Cited on page v.

[182] Lucia Rapanotti, Jon G. Hall, Michael A. Jackson, and Bashar Nuseibeh. Architecture-Driven Problem Decomposition. In Proceedings of the 21st IEEE International Conference on Requirements Engineering, pages 80–89, Los Alamitos, CA, USA, 2004. IEEE.
DOI 10.1109/ICRE.2004.1335666 .
Cited on pages 29 and 149.

[183] Chis Richardson. Microservices Patterns: With examples in Java. Manning Publications, 2018.
URL https://microservices.io/book.
Cited on page 253.

[184] Hugh Robinson, Pat Hall, Fiona Hovenden, and Janet Rachel. Postmodern Software Development. The Computer Journal, 41:363–375, January 1998.
DOI 10.1093/comjnl/41.6.363 .
Cited on pages vi, vii, and 248.

[185] William Robinson, Suzanne Pawlowski, and Vecheslav Volkov. Requirements Interaction Management. ACM Computing Surveys, 35:132–190, June 2003.
DOI 10.1145/857076.857079 .
Cited on pages 171 and 172.

[186] Kenneth S. Rubin and Adele Goldberg. Object behavior analysis. Communications of the ACM, 35(9):48–62, September 1992.
DOI 10.1145/130994.130996 .
Cited on pages 155, 156, and 158.

[187] Jeff Sauro. Better to be approximately right than exactly wrong, April 16, 2016.
URL https://measuringu.com/approx-right/.
Cited on page 270.

[188] Scaled Agile, Inc. Scaled Agile Framework®: SAFe Principles, August 24, 2016.
URL http://www.scaledagileframework.com/safe-lean-agile-principles/.
Cited on page 280.

[189] Scaled Agile, Inc. Scaled agile framework®: Intentional architecture, October 23, 2017.
URL http://www.scaledagileframework.com/architectural-runway/.
Cited on page 150.

[190] Holger Schmidt. A pattern and component based method to develop secure software. Dissertation, University of Duisburg-Essen, Software Engineering (Prof. Dr. Maritta Heisel), 2010.
Cited on page 29.

[191] Holger Schmidt and **Ina Wentzlaff**. Preserving Software Quality Characteristics from Requirements Analysis to Architectural Design. In Volker Gruhn and Flávio Oquendo, editors, Third European Workshop on Software Architecture (EWSA 2006), Revised Selected Papers, volume 4344 of Lecture Notes in Computer Science, pages 189–203, Nantes, France, September 4-5, 2006. Springer.
DOI `10.1007/11966104_14` .
Cited on pages 23, 29, 30, and 149.

[192] Joe Schofield, Alan W. Armemtrout, and Regina M. Trujillo. Function Points, Use Case Points, Story Points — Observations From a Case Study. CrossTalk – the Journal of Defense Software Engineering, 26(3):23–27, 5/6 2013.
`http://www.crosstalkonline.org/storage/flipbooks/2013/201305/index.html`.
Cited on page 42.

[193] Ken Schwaber and Jeff Sutherland. The Scrum Guide™, November 2017.
URL `http://www.scrumguides.org/`.
Cited on pages 6, 26, 174, 176, 177, 178, 182, 183, 246, 247, and 248.

[194] Scopemaster Limited and Colin Hammond. Automated requirements analysis. Patent, (International Application Number: PCT/GB2019/050478, International Publication Number: WO 2019/162676 A2), 08 2019.
URL `https://patentscope.wipo.int/search/en/detail.jsf?docId=WO2019162676`.
Cited on page 175.

[195] Scopemaster Limited and Colin Hammond. ScopeMaster®, 2019.
URL `https://www.scopemaster.com/`.
Cited on page 252.

[196] Scrum.org. Improving the Profession of Software Delivery, 2018.
URL `http://www.scrum.org/Resources/What-is-Scrum`.
Cited on page 6.

[197] Brian Shanblatt. Solved: epic vs story vs task, January 02, 2016.
URL `https://community.atlassian.com/t5/Jira-Core-questions/epic-vs-story-vs-task/qaq-p/204224`.
Cited on page 39.

[198] Mary Shaw and David Garlan. Software Architecture. Perspectives on an Emerging Discipline. Prentice Hall, Eaglewood Cliffs, New Jersey, USA, 1996.
Cited on pages 49, 134, 151, 155, 281, and 298.

[199] M. Shepperd and C. Schofield. Estimating software project effort using analogies. IEEE Transactions on Software Engineering, 23(11):736–743, 1997.
Cited on page 251.

[200] Software Engineering Institute. "Happy Path" Testing. Patterns of Failure: Acquisition Archetypes, 2009.
URL https://www.sei.cmu.edu/library/assets/happy.pdf.
Cited on page 165.

[201] Ian Sommerville. Integrated Requirements Engineering: A Tutorial. IEEE Software, 22(1): 16–23, January 2005.
DOI 10.1109/MS.2005.13 .
Cited on pages 114 and 151.

[202] Ian Sommerville. Construction by Configuration: Challenges for Software Engineering Research and Practice. In 19th Australian Software Engineering Conference (ASWEC 2008), pages 3–12, Perth, Australia, March 25-28, 2008.
DOI 10.1109/ASWEC.2008.4483184 .
Cited on page 114.

[203] Markus Specker and **Ina Wentzlaff**. Exploring Usability Needs by Human-Computer Interaction Patterns. In Marco Winckler, Hilary Johnson, and Philippe Palanque, editors, Proceedings of the 6th International Workshop on Task Models and Diagrams for User Interface Design (TAMODIA 2007), pages 254–260, Toulouse, France, November 7-9, 2007. Springer.
DOI 10.1007/978-3-540-77222-4_20 .
Cited on page 29.

[204] Jeff Sutherland. Scrum: The Art of Doing Twice the Work in Half the Time. Random House Business, 2015.
Cited on pages 2, 3, 6, 10, 15, and 16.

[205] Anneken Tappe. 5 questions with the woman who coined the term 'gray rhino', Jan. 24, 2019.
URL https://www.marketwatch.com/story/what-is-a-gray-rhino-and-why-are-they-so-dangerous-to-investors-5-questions-for-michele-wucker-2019-01-23.
Cited on page 271.

[206] Kevin Tate. Sustainable Software Development : An Agile Perspective. Addison-Wesley Professional, 2005.
Cited on pages 250, 252, 257, and 258.

[207] The SOPHISTs. Requirements Engineering: The SOPHISTs "A short RE Primer", 2016.
URL https://www.sophist.de/fileadmin/user_upload/Bilder_zu_Seiten/Publikationen/Wissen_for_free/RE-Broschuere_Englisch_-_Online.pdf.
Cited on pages 251 and 291.

[208] The Standish Group. Chaos Report, 1995.
URL http://www.cs.nmt.edu/~cs328/reading/Standish.pdf.
Cited on pages 8 and 258.

[209] Total Metrics. What is a Unique Functional Requirement?, 2006.
URL http://www.totalmetrics.com/total-metrics-articles/Function-Points-Unique-Requirements.pdf.
Cited on page 65.

[210] Hugo Troche. Software estimation using pattern analogies, 2004.
URL http://www.developerdotstar.com/mag/articles/troche_patternanalogies.html.
Cited on pages 73 and 251.

[211] Sylvie Trudel and Luigi Buglione. Guideline for Sizing Agile Projects with COSMIC. In 20th International Workshop on Software Measurement (IWSM/MetriKon/Mensura 2010), Vector Consulting Services, Stuttgart, Germany, November 10-12, 2010.
URL http://www.cosmicon.com/portal/public/guideline_for_sizing_agile_projects_with_cosmic_trudel_buglione.pdf.
Cited on page 5.

[212] Sylvie Trudel, Jean-Marc Desharnais, and Jimmy Cloutier. Functional size measurement patterns: A proposed approach. In International Workshop on Software Measurement (IWSM Mensura 2016), October, 5-7, 2016, Berlin, Germany, 2016.
DOI 10.1109/IWSM-Mensura.2016.016 .
Cited on pages 37, 45, 54, 55, 56, 57, 58, 64, and 299.

[213] Bruce Wayne Tuckman. Developmental sequence in small groups. Psychological Bulletin, 63 (6):384–399, 1965.
Cited on page 14.

[214] Portia Tung. The Dream Team Nightmare – Boost Team Productivity Using Agile Techniques. Pragmatic Bookshelf, 2013.
Cited on page 26.

[215] Takuya Uemura, Shinji Kusumoto, and Katsuro Inoue. Function point measurement tool for uml design specification. In Proceedings Sixth International Software Metrics Symposium (Cat. No. PR00403), pages 62–69, 1999.
DOI 10.1109/METRIC.1999.809727 .
Cited on page 73.

[216] Bill Wake. INVEST in Good Stories, and SMART Tasks, August 17, 2003.
URL https://xp123.com/articles/invest-in-good-stories-and-smart-tasks/.
Cited on page 39.

[217] Paul T. Ward and Stephen J. Mellor. Structured Development for Real-Time Systems, volume 1: Introduction and Tools. Yourdon Press, 1985.
Cited on pages 25, 113, 115, 116, 117, 118, 159, and 299.

[218] Paul T. Ward and Stephen J. Mellor. Structured Development for Real-Time Systems, volume 2: Essential Modeling Techniques. Yourdon Press, 1985.
Cited on pages 122, 123, and 134.

[219] Paul T. Ward and Stephen J. Mellor. Structured Development for Real-Time Systems, volume 3: Implementation Modeling Techniques. Yourdon Press, 1985.
Cited on pages 115, 117, 118, 119, 120, 134, 135, and 151.

[220] Kelly Waters. The Value of Stable Teams, 2011.
URL http://www.allaboutagile.com/the-value-of-stable-teams/.
Cited on pages 14 and 43.

[221] Kelly Waters. All about Agile: Agile Management Made Easy! Createspace, 2012.
URL https://www.101ways.com/blog/.
Cited on pages 4, 5, 7, 39, and 108.

[222] Gregory H. Watson. The Benchmarking Workbook: Adapting Best Practices for Performance Improvement. Productivity Press, 1992.
Cited on page 193.

[223] **Ina Wentzlaff**. Establishing a Requirements Baseline by Functional Size Measurement Patterns. In First International Workshop on Requirements Prioritization and Enactment (PrioRE'17), CEUR Joint Proceedings of REFSQ 2017 Workshops co-located with the 23nd International Conference on Requirements Engineering: Foundation for Software Quality (REFSQ 2017), Essen, Germany, February 27, 2017.
URL http://ceur-ws.org/Vol-1796/priore-paper-1.pdf.
Cited on pages 23, 28, 40, and 299.

[224] **Ina Wentzlaff** and Markus Specker. Pattern-Based Development of User-Friendly Web Applications. In Workshop Proceedings of the Sixth International Conference on Web Engineering, Palo Alto, California, USA, July 11-14, 2006. ACM.
DOI 10.1145/1149993.1149996 .
Cited on pages 29 and 70.

[225] Karl E. Wiegers. More About Software Requirements: Thorny Issues and Practical Advice. Microsoft Press, Redmond, WA, USA, 2005.
Cited on pages 5, 15, 16, 17, and 20.

[226] Karl E. Wiegers and Joy Beatty. Software Requirements. Developer Best Practices. Microsoft Press, Redmond, WA, USA, 3rd edition, 2015.
Cited on pages 5, 6, 26, and 73.

[227] Niklaus Wirth. A Plea for Lean Software. IEEE Computer, 28(2):64–68, 1995.
DOI 10.1109/2.348001 .
Cited on pages 11 and 26.

[228] Dave Wolber and Neil Calder, editors. ICWE '06: Workshop Proceedings of the Sixth International Conference on Web Engineering, New York, NY, USA, 2006. ACM.
Cited on page 70.

[229] Sherif M. Yacoub and Hany H. Ammar. Pattern-oriented analysis and design (POAD): a structural composition approach to glue design patterns. In Proceedings of the 34th International Conference on Technology of Object-Oriented Languages and Systems - TOOLS 34, pages 273–282, August 2000.
DOI 10.1109/TOOLS.2000.868978 .
Cited on page 114.

[230] Sherif M. Yacoub and Hany H. Ammar. Pattern-Oriented Analysis and Design: Composing Patterns to Design Software Systems. Addison-Wesley Professional, 2004.
Cited on page 113.

[231] Edward Yourdon. Modern Structured Analysis. Yourdon Press, 1989.
Cited on page 115.

Notice: Product or corporate names may be trademarks or registered trademarks, and are used only for identification and explanation without intent to infringe.

What have we learned?
Even a snail can do a Sprint.

DOI: 10.17185/duepublico/74971
URN: urn:nbn:de:hbz:464-20211119-145934-6