



PADERBORN UNIVERSITY
The University for the Information Society

**Project-Specific Software
Engineering Methods**
Composition, Enactment, and Quality Assurance

Masud Fazal-Baqaie

Faculty of Computer Science, Electrical Engineering, and
Mathematics
Paderborn University
Dissertation submitted in partial fulfillment
of the requirements for the degree of
Doktor der Naturwissenschaften (Dr. rer. nat.)

September 2016

I dedicate this thesis to all young people at the Hindu Kush who pursue knowledge and wisdom to overcome their boundaries.

Acknowledgements

This thesis has been made possible by support of various people. In the following, I would like to express my gratitude and thank them.

First, I would like to thank my advisor Prof. Dr. Gregor Engels. Gregor, thank you for giving me the opportunity to pursue my degree in your research group and for advising me with critical feedback and motivating words. I am especially grateful for your continuous support in phases of intensive project work and during the final phase of writing up the thesis. To a very large extent my research is based on industrial experience that I gathered in projects within the s-lab – Software Quality Lab. Thus, I would also like to thank the managing director of s-lab and co-author of several of my papers, Dr. Stefan Sauer. I also thank the external co-reviewer of my thesis Prof. Dr. Marco Kuhrmann. Thank you Marco for the fruitful discussions on various venues and your thorough review and improvement suggestions on drafts of this thesis. In addition, I would like express my gratitude to Prof. Dr. Eric Bodden and Prof. Dr. Dennis Kundisch for being members of my doctoral committee.

During my research, I collaborated with various people and I would like to thank all of them for the joint work on publications or their critical reviews and feedback. I would like to mention a couple of them explicitly. First, I would like to thank Dr. Markus Luckey for being the co-author of my first paper on my thesis topic. I would like to thank Dr. Marvin Grieger for our productive collaboration on several paper projects and for the sparring during writing up our theses. I also thank Dr. Baris Güldali and Prof. Dr. Christian Gerth for our collaboration. Dennis Wolters provided me with valuable feedback on several papers and presentations, thank you for that. In addition, I would like to acknowledge the following people that I advised. Thank you for your contributions to my tooling with your master theses and your work as student assistants: Subramanya Gurusamy, Frank Kluthe, Karthik Neela, Karimuddin Cuddapah Shaik, Daniel Siebert.

I would also like to express my gratitude to the various project partners that allowed me to gather practical insights. In particular, I thank HJP Consulting GmbH and S&N AG for the opportunity to work abroad.

This thesis is also influenced by the discussions within the German-speaking community of the GI Special Interest Group on Process Models for the Development of Business Application Systems (GI-Fachgruppe Vorgehensmodelle für die betriebliche Anwendungsentwicklung). I would like to thank all former and current committee members for contributing to this community.

Fortunately, the last couple of years in pursuit of my degree were not only characterized by work. I would like to thank the former colleagues for the warm and friendly environment of the Database and Information Systems Research Group and the s-lab, especially my office mates Dr. Christian Soltenborn, Dr. Hendrik Schreiber, and Enes Yigitbas. I also thank Ivan Jovanovic and Dr. Stefan Grösbrink for our basketball games and Mirko Rose for our discussions on exotic travels. Thanks to Dr. Henning Wachsmuth, who showed his support by joining my defense.

I would also like to thank my friends outside of work – thank you for helping me to forget about work from time to time. I thank my former flatmates (Kircherweg, Königskinder, M7), the Paderborn salsa and climbing communities, and especially: Navneet Bhalla, Ph.D.; Olga Käthler; Regina Dirks; Christina Rieke; Sarah Altmann; Dr. Darko Jus; Danka Dasic; Simon Willmes; Christin Kirchhubel, Ph.D.; Dr. Jörg Clobes; Stephan Parzefall; Tine Niederhacker; Vebe Neamen. Thank you Matthias Becker for being my best Bro and for being there, when I am in need for an open ear.

Finally, I thank my parents and sisters for their unconditional love and support. I thank Irene Palnau for her patience and her sacrifices, when I was working on my thesis on evenings and on weekends. Rain-soaked hiking tours turn into light-hearted singing sessions with you, Irene.

Abstract

Software engineering methods describe structured, repeatable best practice approaches for the engineering of software systems. The project team of a software project enacts a method and applies the described activities. As methods are superior to ad-hoc build and fix approaches, they benefit the creation of high-quality software. However, for the efficient use of methods, first, they need to be based on state of the practice method content, second, they need to be tailored to the project context, and third, they need to be enacted as prescribed. Otherwise, outdated, unsuitable, or wrongly enacted methods can impede the creation of the software system. While other approaches focus on supporting some of these aspects, our approach is a holistic tool-supported approach that covers all of them. It allows creating formally defined composition-based method models. First, method models are composed from formal building blocks that represent method content and are stored in an extensible, updatable repository. Second, they are composed specifically for a project and tailored to its characteristics. Here the novel notion of method patterns is used to guide the composition process. Third, their correct enactment is supported with a process engine. Our proof-of-concept implementation demonstrates the feasibility of the approach. It provides tooling to define building blocks, to compose them to method models consistently, and to execute them with standard process engines.

Zusammenfassung

Softwareentwicklungsmethoden beschreiben Best-Practice-Ansätze für die Entwicklung von Softwaresystemen. Damit sind Methoden einfachen Ad-Hoc-Ansätzen überlegen und ihr Einsatz unterstützt die Entwicklung von hochqualitativer Software. Jedoch erfordert der effektive Einsatz von Methoden, drei Dinge: Erstens müssen Methoden auf aktuellen Methodeninhalten basieren, zweitens müssen sie auf den Projektkontext angepasst werden und drittens müssen sie wie vorgeschrieben von dem Projektteam angewendet werden. Ansonsten gefährden veraltete, unangepasste oder falsch angewendete Methoden den Projekterfolg. Während andere Ansätze nur einige dieser Aspekte abdecken, präsentieren wir einen umfassenden, werkzeuggestützten Ansatz, der alle Aspekte des Managements von Softwareentwicklungsmethoden abdeckt. Unser Ansatz ermöglicht die Erstellung von formalen, kompositions-basierten Methodenmodellen. Erstens werden Methodenmodelle aus formalen Methodenbausteinen zusammengesetzt. Diese repräsentieren, aktuelle Methodeninhalte und werden in einer aktualisierbaren Methodenbasis gehalten. Zweitens werden Methodenmodelle projektspezifisch und kontextbasiert komponiert. Drittens wird ihre korrekte Anwendung durch den Einsatz einer Process-Engine sichergestellt. Unsere Proof-Of-Concept-Implementierung demonstriert die Machbarkeit unseres Ansatzes und stellt Werkzeugunterstützung für die Definition von Methodenbausteinen, die konsistente Methodenmodellkomposition und die Ausführung mit Standard-Process-Engines zur Verfügung.

Table of contents

List of figures	xv
List of tables	xix
1 Introduction	1
1.1 Motivation and Problem Statement	1
1.2 Solution Overview and Research Contributions	6
1.2.1 Tasks of the Senior Method Engineer	7
1.2.2 Tasks of the Project Method Engineer	8
1.2.3 Tasks of the Project Team	10
1.3 Publication Overview	11
1.4 Structure of the Thesis	12
2 Background	15
2.1 Foundations and Terminology	16
2.1.1 Application Development with Software Engineering Methods	16
2.1.2 Situational Method Engineering	21
2.1.3 Executable Process Description Languages	25
2.2 Solution Requirements and State of the Art	26
2.2.1 Requirements for a Holistic Solution	27
2.2.2 Evaluation of Existing Approaches	31
2.3 Summary	36
3 Solution Overview	37
3.1 Overview of the MESP Approach	37
3.1.1 Overview of MESP Roles	38

3.1.2	Overview of MESP Work Products	41
3.1.3	Overview of MESP Tools	43
3.1.4	Integrated Overview of MESP Solution	43
3.2	End-to-End Example	46
3.2.1	Method Content Definition	46
3.2.2	Method Tailoring	56
3.2.3	Method Enactment	64
3.3	Summary	68
4	Method Content Definition	71
4.1	Requirements and Related Work	72
4.1.1	Requirements	72
4.1.2	Related Work	73
4.2	Extract Reusable Method Content	75
4.2.1	Extraction from Methods Described in Literature	75
4.2.2	Extraction from the Daily Practice of Organizations	79
4.3	Define Basic Elements	83
4.3.1	Definition of Basic Method Elements	84
4.3.2	Definition of Basic Characterization Elements	87
4.4	Define Method Services & Method Patterns	89
4.4.1	Definition of Method Services	91
4.4.2	Definition of Method Patterns	94
4.5	Summary	101
5	Method Tailoring	105
5.1	Requirements and Related Work	106
5.1.1	Requirements	106
5.1.2	Related Work	108
5.2	Characterize Project	108
5.2.1	Meta-Classes	109
5.2.2	Usage	110
5.3	Compose Project-Specific Method	111
5.3.1	Identifying Suitable Method Building Blocks	112
5.3.2	Specification of Methods	114
5.4	Assure Quality of Method	122
5.4.1	Quality Model	125
5.4.2	Automated Quality Assurance Framework	134
5.4.3	Usage	139
5.5	Initialize Method	141
5.5.1	Transformation, Deployment & Configuration	143
5.5.2	Usage	148
5.6	Summary	151

6	Method Enactment	153
6.1	Requirements and Related Work	154
6.1.1	Requirements	154
6.1.2	Related Work	155
6.2	Coordinate Activities	156
6.3	Perform Tasks	160
6.4	Reflect Method	163
6.5	Summary	165
7	Proof of Concept Implementation	167
7.1	Tool Implementation	168
7.1.1	Method Content Definition	168
7.1.2	Method Tailoring	170
7.1.3	Method Enactment	175
7.2	Method Composition	176
7.2.1	Case Study: Certification Issuance Process	176
7.2.2	Experiment: Scalability Analysis	183
7.3	Summary	187
8	Conclusions and Outlook	189
8.1	Contribution Summary	189
8.2	Fulfillment of Requirements	192
8.3	Outlook on Future Work	195
	References	199
	Acronyms	215

List of figures

1.1	The three conceptual layers of software engineering method management	3
1.2	Overview of the MESP framework and our contributions	6
1.3	Overview of previously published material	11
2.1	The concepts related to software engineering methods	17
2.2	Trade-off between effort and flexibility of different groups of method engineering approaches	22
2.3	Tasks of assembly-based method engineering	24
2.4	The relationship between software engineering method management layers, our solution requirements (SRs), and the common requirements in [Ell+11]	29
3.1	The aspects included in our MESP solution	38
3.2	Overview of the MESP Tasks	39
3.3	Overview of central MESP Work Products	42
3.4	Overview of MESP Tool Support	44
3.5	An Overview of the MESP Solution Framework	45
3.6	The gateways of the V-Modell XT method	47
3.7	The decision gate "System designed" of V-Modell XT	47
3.8	Illustration of the process flow of Scrum	48
3.9	Decription of the task "refine the architecture" of OpenUP	49
3.10	Basic elements created by the senior method engineer	50
3.11	A method service derived from OpenUP referencing basic elements	52
3.12	A method pattern based on the Sprint from the Scrum method	54
3.13	The object model of the method pattern based on the Sprint from the Scrum method	54

3.14	The method pattern for the V-Modell XT is derived by creating constrained scopes for each decision gate	55
3.15	Constraint scopes that reflect the decision gate “System Designed” of V-Modell XT	55
3.16	Relationship of the Work Products for Method Content Definition	56
3.17	The MESP tools to define method content	57
3.18	The characterization of a project with basic elements from the method repository	58
3.19	Two method patterns combined to one pattern	59
3.20	A partial method model with method services	60
3.21	A partial method model with additional control and data flow	61
3.22	A correctly specified partial method model	62
3.23	Configuration Interface of the BPEL engine to set up users and roles	63
3.24	Relationship of the MESP Work Products for Method Tailoring	64
3.25	The MESP tools for Method Tailoring	65
3.26	The coordination of activities via workflow tasks	66
3.27	A workflow task for the method service Refine the Architecture	67
3.28	Relationship of the MESP Work Products for Method Enactment	69
3.29	The MESP tools for Method Enactment	69
4.1	Extraction of method content from methods described in literature	76
4.2	Extraction of method content from the daily practice of organizations	79
4.3	Excerpt of package structure of MESP meta-model	84
4.4	Illustration of division between Method Content and Process in SPEM (adopted from [OMGo8])	85
4.5	The basic method elements of our meta-model package BasicMethodElements	86
4.6	The basic characterization elements of our meta-model package BasicCharacterizationElements	88
4.7	Excerpt of the package structure for method building blocks of the MESP meta-model	91
4.8	The method service related meta-classes of our meta-model package MethodService	93
4.9	A method service derived from OpenUP	94
4.10	The method pattern related meta-classes of our meta-model package MethodPattern	95
4.11	The condition related meta-classes of our meta-model package patternElements	96
4.12	The service characterization related meta-classes of our meta-model package patternElements	98
4.13	The work product characterization related meta-classes of our meta-model package patternElements	100

4.14	The object model of the method pattern derived from Scrum shown in Figure 3.12	102
5.1	The project characteristics related meta-classes of our meta-model package ProjectMethod	110
5.2	The control-flow related method elements of our meta-model package ProjectMethod	115
5.3	The data-flow related method elements of our meta-model package ProjectMethod	118
5.4	The object diagram for the composed method patterns of Figure 3.19	120
5.5	The object diagram for the <i>Develop</i> constrained scope of Figure 3.22	121
5.6	The object diagram with the data flow specification between two method service descriptors according to Figure 3.22	123
5.7	MESP's quality model for software engineering method models . . .	126
5.8	Overview of the quality assurance framework	135
5.9	OCL expression to find elements that violate precedence consistency	136
5.10	OCL definition of the helper function getNested()	137
5.11	A Java code snippet for the translation of quantifiers to OCL	138
5.12	A partial method model with quality issues	139
5.13	The generated OCL expressions for the middle constraint scope descriptor of Figure 5.12	140
5.14	The execution of a BPEL/BPEL4People process model	144
5.15	Snippet of the resulting BPEL process for the end-to-end example .	149
5.16	The configuration interface of the BPEL engine to configure role assignment	150
5.17	View of the project repository with uploaded WorkProduct	150
6.1	A partial example method model based on the end-to-end-example	157
6.2	The process model for the method model of Figure 6.1	158
6.3	Task management view of architect Bob with a workflow task ready	159
6.4	A workflow task to decide about a further run of an iteration	160
6.5	The Architecture Notebook uploaded to the project repository by Bob	161
6.6	The workflow task for Envision the Architecture	162
6.7	The workflow task for Refine the Architecture	164
6.8	The change log of the project repository	165
7.1	Definition of a Task using the EPF Composer	169
7.2	Defining a Method Pattern using the Tree-based EMF Editor	169
7.3	The end-to-end example in the tree-based editor	171
7.4	Composition of a process in the extended BPEL Designer	171
7.5	A composed method model in the customized BPEL Designer	172
7.6	The end-to-end example in the Sirius-based editor	173

7.7	An issue reported in the Problems View and visualized in the Sirius Editor	174
7.8	The context menu of the tree-based editor showing the command to transform a method model to an BPEL process model	175
7.9	The captured process for the certification of ePassports	177
7.10	Excerpt of the captured process for the certification of ePassports . .	178
7.11	The work products defined based on the BPMN process diagram . .	179
7.12	The tasks defined based on the BPMN process diagram	179
7.13	The roles defined based on the BPMN process diagram	180
7.14	The derived method services for the case study	180
7.15	Details of the method service Perform ICAO Test	181
7.16	The composed Method for the certification of ePassports	182
8.1	Contribution Overview of our thesis	190

List of tables

2.1	Common requirements for software engineering method management solutions without refinement regarding software engineering management layers or assembly-based method engineering (adapted from [Ell+11])	28
2.2	Evaluation of existing tool-supported approaches for assembly-based method engineering	35
4.1	Method definition requirements and the affected MESP tasks	74
5.1	Method tailoring requirements and the affected MESP tasks	107
5.2	Evaluation Criteria for quality models (adopted from [GK09])	133
5.3	Refined requirements for the analysis framework	134
5.4	The mapping of MESP and BPEL/BPEL4People concepts	145
6.1	Method Enactment Requirements and the affected MESP tasks	155
7.1	The data sets of process models for the scalability analysis	184
7.2	Results of the scalability analysis of the Consistency Checker	185
7.3	Results of the scalability analysis of the MESP2BPEL Transformer	186

CHAPTER 1

Introduction

In this chapter, we explain the motivation and problem statement for this thesis. We point out our solution approach and contributions. We present an overview of publications published in the context of this thesis and provide a structural overview of the thesis.

1.1 Motivation and Problem Statement

In *software engineering*¹, similar to other engineering disciplines, a structured approach to the development of the software systems is seen as a key factor for the resulting product and service quality [FRO03]. *Software development processes* describe such a structured, repeatable best practice. We thereby understand software development processes to comprise project management aspects, organizational aspects, and methodological aspects. Project management aspects denote the scheduling and allocation of available resources, organizational aspects denote the definition of organizational units and allocation of responsibilities, and methodological aspects denote how a software system should be developed systematically.

In the following, we focus on the method aspect of software development processes and use therefore the term *software engineering method*, in short, *method*. Software engineering methods prescribe who should carry out which *activities* in what order and thereby describe how *artifacts* should be derived from one another. In general, software engineering methods cover the typical lifecycle phases of a *software engineering endeavor*, e.g., a *software project*. These are *software specification*,

¹Key terms are defined in Section 2.1.1 and Section 2.1.2.

software design and implementation, software validation, and software evolution. Normally, these phases are not carried out purely sequentially, but earlier phases are revisited regularly [Som11]. In most of the cases, software engineering methods are described textually together with some informal visualizations of the high-level process of activities or phases. If a method is described formally, e.g., by using a *meta-model*, we call it a *software engineering method model* or method model. There exist both methods for general-purpose software engineering and methods that are specialized to a specific discipline or domain, e.g., Testing [SLS14] or Automotive [Hoe08].

In this thesis, we focus on software engineering methods for software projects. That is, we do not discuss, organization-wide software engineering methods, often referred to as reference processes.

The use of software engineering methods benefits a software project, primarily in two ways [CKO92]: First, it helps the group of involved people to coordinate the various activities and to establish a common vocabulary of how to plan, develop, and maintain a software system. Second, software engineering methods provide guidance for the individual on how to carry out a specific activity.

During the past decades, several software engineering methods have been published as reference and best-practice examples based on the experience in different software engineering endeavors. For example, among others, the methods Quasar Enterprise [Engo8] and Rational Unified Process (RUP) [Kru99] were initially created by their respective IT companies for their own software engineering projects. Later, they were published as a reference for the public.

While using a software engineering method is in general superior to an ad-hoc build-and-fix approach, there is nothing like an one-size-fits-all software engineering method [Gla04]. Instead, a method that is suitable and beneficial in one context can lead to quality issues in a different context. For example, several publications describe how methods are used successfully in smaller, co-located settings (e.g. [BT03]), while the same methods cause issues in a distributed, global software engineering setting (e.g. [Ram+06],[CR06],[FSH15]). Therefore, the reusability of methods is limited and they need to be customized to the context of the software engineering endeavor – they need to be *tailored* to the *situation* [Hen+14].

Looking at tailoring from a conceptual point of view, we can differentiate three layers of *software engineering method management*. We define this *software engineering method management hierarchy* consisting of method content definition, method tailoring, and method enactment as follows (adapted from [Heno6], cf. Fig. 1.1): On the upper *method content definition* layer of the hierarchy, reusable method knowledge is defined by experienced experts, for example, as published methods like RUP and Quasar Enterprise. On the middle *method tailoring* layer, these methods need to be tailored to the situation, for example, by the process manager of an organization or a project manager. On the lowest *method enactment* layer, the

tailored method is then to be *enacted* by the organization or *project team* by following the method. Software engineering method management is the coordination of efforts to accomplish these objectives.

The method-related content created on one layer influences the content on the layer below: the amount and form of available method content created on the topmost layer influences how easy the method can be tailored to the situation on the middle layer. For example, a textual representation defined on the top layer is more difficult to tailor in a consistent manner than a formal representation. If alternative activities are defined on the top layer, this additional choice eases the tailoring. The content and form of the tailored method on the middle layer, in turn, directly influences how easy the people can apply and follow the method. For example, a textual representation is more ambiguous than a formal one and a method that includes more detailed *guidances* and *task descriptions* is easier to enact.

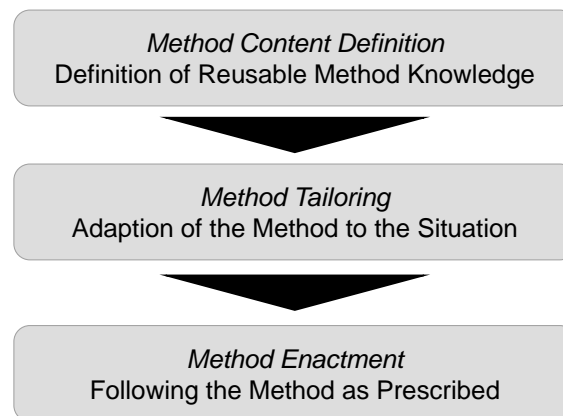


Fig. 1.1 The three conceptual layers of software engineering method management

Challenges of Software Engineering Method Management

Each layer of the software engineering method management hierarchy poses its individual challenges. In the following, we discuss the challenges for each layer.

Regarding the definition of method content, it is difficult to update the content of methods incrementally in order to reflect current trends, best practices, and lessons learned [KF15]. For example, as the *agile methodology* [Mey14] is becoming more and more popular, many creators of methods face the challenge to integrate agile principles into their methods [VB15],[BT03]. Consequently, the users of these methods cannot make use of the advances and improvements before new versions of the methods are released. One major obstacle for updating methods is that they are typically described in a textual manner, what makes it difficult to clearly scope the area to improve and to assess the implications for the unchanged parts [WBV07].

Regarding the tailoring of methods, methods are sometimes defined in such a way that they do not especially address the need for tailoring or they even disallow it [Hen+14]. However, many comprehensive methods, e.g., RUP, Quasar Enterprise, V-Modell XT [KTF11], or MFESA [Firo9] acknowledge the need to tailor a method to a situation. They are published as *software engineering method frameworks* (termed *process framework* in [KFS13a]) that do not describe a uniform method to be used in every project, but support a certain degree of tailoring. Therefore, they require that as a preliminary step, the method has to be created by selecting and adjusting from the provided content. Agile methods, e.g., [SS13], [Bec00], [Coc05] follow a different approach to achieve the tailoring of the software engineering method to the situation. They focus on the lightweight description of some mandatory core *roles*, activities and artifacts, while the rest is left to be defined and adapted by the project team based on its experience and the current circumstances of the situation. For example, the popular and widely adopted software engineering method Scrum² [SS13] includes a periodical activity called *Retrospective*. This activity is dedicated to perform required changes (method tailoring) to the method itself. The actual software engineering activities (method content definition), however, are described in the most generic way possible.

Irrespective of the underlying method, tailoring a method in a meaningful and consistent way is a non-trivial task that needs to be performed by experts [KF15],[Hei+10]. One of the reasons for this is that many methods have no underlying formalism at all and are described purely in natural language, e.g., Scrum, Quasar Enterprise, or the Crystal Family [Coc98; Coc02; Coc05]. Therefore, ensuring the consistency of a method is more difficult. Both RUP and V-Model XT evolved to a model-based approach, where the tailored methods are instances of a meta-model that describes the rules for *consistent* method models. However, there is only very limited support and guidance on how to tailor a method model with respect to the situation, e.g., whether an activity should be added or replaced. Even though the meta-model of RUP evolved to the industry standard Software & Systems Process Engineering Meta-Model Version 2.0 (SPEM) [OMGo8] in 2008, the *modeling language* is still not commonly used. In addition, the language itself has been criticized, because it does not allow specifying an *executable method model* [Ell+10], what makes it more difficult to follow the method properly.

Regarding the enactment of methods, it has to be ensured that how the work is actually carried out corresponds to the defined method, otherwise there is the risk of missing out important activities or doing them wrongly [Ost87]. For example, in global software projects it is challenging to maintain sufficient overview and to coordinate the activities according to the defined method [NC13]. Typically, there is only generic tool-support for tailored methods available as these are often

²Scrum focuses on the project management aspects of software development processes, but is typically complemented with agile principles.

described in natural language and tool support cannot be derived automatically. While Application Lifecycle Management (ALM) suites [KV09] offer tool support of the implemented general-purpose methods, they typically do not provide guidance for the tailoring of methods.

Problem Statement

Acknowledging the difficulties of software engineering method management, the research area *situational method engineering*³ is dedicated to the provisioning of situation-specific methods [Hen+14]. Here, several approaches to provide and manage method content, to derive situational software engineering methods, and to enact them were proposed in the past. Each approach varies in its level of support for the three software engineering method management levels. At present, existing approaches do not support all three levels sufficiently in one comprehensive solution. Approaches like [Har97; GP01; Ral04; KLR96; Spi15] lack in particular a formal foundation that allows checking tailored methods for consistency and that allows to execute them. Other approaches like [Ell+11; Ben+07; Wis+00] offer a modeling language that allows to model executable method models, but do not sufficiently support the tailoring of methods and the definition of reusable method content.

In summary, to leverage the advantages of software engineering method management, the entire lifecycle of software engineering method models needs to be addressed. To that extend, a suitable solution for the definition of method content, its tailoring to a situation, and its enactment is required. Such a solution needs to address the following challenges:

- It must enable updates of method content based on new trends, best practices, and lessons learned
- It must enable the creation of consistent software engineering method models for specific situations based on defined method content
- It must enable the proper enactment of the tailored method models according to their definition

In this thesis, we introduce a comprehensive solution for software engineering method management that tackles the described challenges. Our solution is based on research and industrial project work performed within the s-lab – Software Quality Lab of the University of Paderborn in joint projects with HJP Consulting GmbH and other industrial partners. The research has been partly funded by the German Federal Ministry for Economic Affairs and Energy (Bundesministerium

³discussed in detail in Section 2.1.2

für Wirtschaft und Energie). An overview of our proposed solution together with our research contributions is presented next.

1.2 Solution Overview and Research Contributions

An overview of our solution for software engineering method management, addressing the described challenges, is provided in Fig. 1.2. Our solution framework addresses all major tasks on the three layers of software engineering method management. An overview of our solution is presented in Chapter 3. The tasks on each layer are then described in the chapters 4-6 of this thesis. A proof of concept implementation with a case study and an experiment is described in Chapter 7.

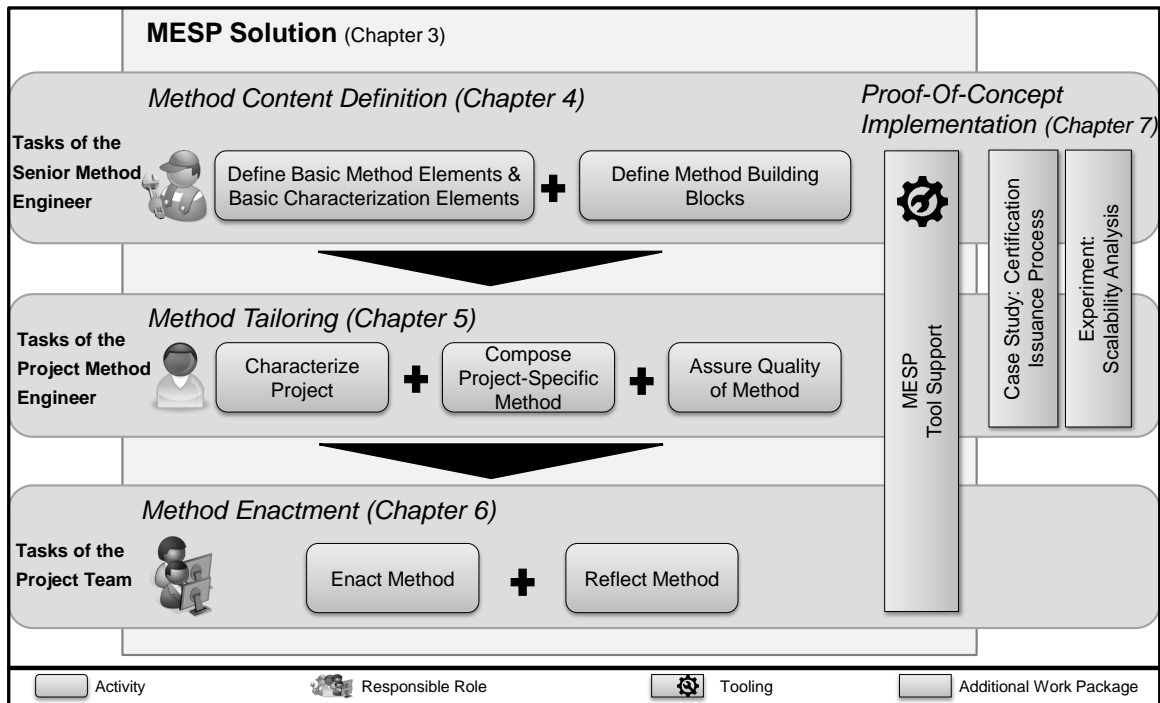


Fig. 1.2 Overview of the MESP framework and our contributions

Our solution for software engineering method management is called Method Engineering with Method Services and Method Patterns (MESP). *Method services* and *method patterns* are the two types of *method building blocks* that are fundamental to our solution. We differentiate three roles, where each role is responsible for the tasks of one layer of the software engineering method management hierarchy, because each layer requires a different level of knowledge and experience.

Senior method engineers are responsible for defining method content, based on lessons learned or information sources like methods described in literature. They

model *basic elements* and on top of that the actual method building blocks, which can be used to create situation-specific software engineering method models.

Project method engineers then can choose suitable method building blocks for their project and compose them to a tailored method model for their project.

The *project team* follows the method to create the software in their project. The composed method model is *executed* with a *process engine* (termed *workflow management software* in [AHO2, p. 148]) that coordinates the activities of the project team members, provides them with guidance on the pending tasks, and thus ensures that they enact the method as prescribed.

As part of our end-to-end solution, we defined the necessary meta-models and algorithms to model method building blocks, to model composed method models, to ensure their consistency, and to execute them with a process engine. Tool support for our solution is implemented as a prototype that supports most of the tasks described in the solution and that demonstrates the applicability of our approach.

In the following, we briefly describe the tasks for each role and layer and point out our research contributions.

1.2.1 Tasks of the Senior Method Engineer

The senior method engineer is responsible for defining and maintaining the method building blocks that form the pool of available building blocks for the creation of situation-specific method models for specific projects.

These building blocks need to reflect the state of the art of method engineering, e.g., best practices and lessons learned, including content from agile and plan-driven methods, in order to allow for composing state of the art situation-specific method models. In order to do that, building blocks should be composable and capable of reflecting not only single activities, but also composite activities.

In order to reduce the tailoring effort for the project method engineer, building blocks should offer meaningful sequences and orderings of activities. In addition, the building blocks need also to be characterized in terms of their suitability for different situations.

In order to allow for the execution of composed method models with a process engine, the method building blocks must be formalized and executable.

We define two kinds of building block types to capture composable method content. Method services basically describe atomic or composite activities, e.g., to capture the task in a method to *Identify Stakeholders* as part of the analysis lifecycle phase [Bal07],[RR13, p. 54]. The notion of method services is influenced by the service-oriented paradigm [Erlo9, p. 70]. To enable interoperability and composition, each method service contains a uniform interface description.

In addition, we introduce the novel notion of method patterns that allows to capture best practices for the ordering of activities on an activity-independent level. For

example, it allows capturing the idea of an agile sprint loop [SS13],[CH05],[Mey14] without fixing the involved activities. These method patterns serve as additional guidance for the project method engineer during method composition. We define a meta-model that allows her to model these building blocks in a formal and uniform manner.

Moreover, each building block is characterized with its suitability for specific project characteristics, e.g., *small development team* or *low stakeholder participation*. This helps less experienced project method engineers to identify suitable method building blocks for their project. We extend our meta-model to allow the senior method engineer to formalize these *characterizations* with so-called *situational factors* [Bec00],[CO12].

In order to ensure the executability of method building blocks, we include an executable process description language into our meta-model to describe the control flow within composite method building blocks.

The main contributions are as follows:

- The definition of a senior method engineer role and its tasks to offer method building blocks
- The description of two ways to determine and extract reusable method content, first, from existing methods described in literature, and second, from the daily practice of organizations
- The definition of a method service, a reusable, compositional, interoperable, and executable unit of method based on the service-oriented paradigm
- The definition of a method pattern, a means to capture abstract orderings of activities as a guidance for the project method engineer
- A formalization in terms of a meta-model for the definition of method services, method patterns, and underlying basic method model elements like roles and work products (artifacts). Furthermore, a formalization of situational factors for the characterization of method services and method patterns

1.2.2 Tasks of the Project Method Engineer

The project method engineer is responsible for composing a software engineering method model with respect to a specific project by using the available method building blocks. Thereby, she can make use of the expertise of senior method engineers, who created the building blocks.

In order to compose a method model that is suitable for the project, she has to be able to characterize the project situation in order to pick suitable method building blocks based on their situational factors.

In order to ensure that the composed method model can be executed with a process engine later by the project team, the selected method patterns and method services need to be composed properly. It follows that the composition language must be executable and define *control flow* as well as *data flow* [GJM03, pp. 179;171] between method services. In addition, it has to be ensured that the composed method model is consistent, so that no issues occur later during enactment. The project method engineer is also responsible for preparing the method enactment, e.g., by assigning project team members to roles defined in the method.

We define a formal project method interface that allows a project method engineer to characterize a project prior to a method composition. The project method interface uses the same basic elements that are also part of the method service interfaces.

We implement a transformation from our method models to *process models* formalized with the Business Process Execution Language (BPEL) [OAS07] and BPEL4People [OAS10], offering a generic graphical user interface (GUI) for each task.

In addition, we define a formal composition language that reuses the executable process description language defined for method building blocks and that includes control as well as data flow.

To ensure that the composed method is consistent, we define an extensible quality assurance framework and implement quality assurance rules that check for quality issues like missing building blocks, contradictions in control and data flow, and inconsistencies between the method and the project method interface.

Our solution allows executing the composed method models with a standard process engine (termed *workflow management software* in [AH02, p. 148]). We implement a transformation algorithm that transforms a method model into a standard-conforming process model that can then be executed with the process engine. As part of the initialization of the method enactment, the project method engineer invokes this transformation. She then uses the facilities of the process engine to deploy and initialize the model, e.g., by assigning project members to roles that are defined in the method.

The main contributions are as follows:

- The definition of a project method engineer role and its tasks to compose a method for a situation
- The definition of a project method interface to formalize the characterization of a situation
- The definition of an executable composition language that includes the specification of control and data flow
- A quality assurance framework for the analysis of composed methods together with a set of quality assurance rules to check method models for quality issues

- A transformation algorithm to transform method models to BPEL/BPEL4People conformant process models allowing to execute methods with a standard process engine including a generic GUI that allows project team members to interface with the process model during method enactment

1.2.3 Tasks of the Project Team

The project team is responsible for enacting the composed method model.

In order to ensure that the project team members perform the activities in the same order that is described in the composed method model, they need to coordinate their tasks and inform each other about the progress.

In order to ensure that an assigned project team member performs the assigned task with the right input work products and the way it was prescribed, the team member needs easy access to the description of the task and the location of input work products.

In order to reflect about the method enactment for later improvement of method building blocks, the project team needs to collect information about method-related issues.

As mentioned in the previous section, we implement a transformation from our method models to models formalized with BPEL and BPEL4People. These process models can be executed with a standard BPEL process engine (*BPEL engine*), which ensures that the method is enacted as specified. The BPEL engine ensures that the activities are performed in the right order by coordinating the order of tasks that it assigns to individuals. In addition, it shows information about the current activities.

Our transformation creates a GUI that is used for each task and allows the project team member to inform herself about it. It shows a description about the current task that originate from the method building blocks used in the composed model. In addition, it shows the location of input work products and allows specifying the location of the created work products as part of the task.

The BPEL engine logs data about the execution of the process model that can be used when reflecting about method-related issues.

The main contributions are as follows:

- The definition of a project team role and its task to enact a method
- The mapping of method model concepts to BPEL/BPEL4 in order to support the method enactment with the process model execution including the GUI for project team members
- The description of a way to reflect the method enactment in order to support the senior method engineer in improving method services and method patterns

1.3 Publication Overview

Our solution for the method engineering method management presented in this thesis has been influenced by research and industrial project work in the context of method engineering. The research has been partly conducted in a joint project with HJP Consulting GmbH founded by the Central Innovation Programme for Small and Medium Enterprises of the German Federal Ministry for Economic Affairs and Energy (Zentrales Innovationsprogramm Mittelstand des Bundesministerium für Wirtschaft und Energie). The industrial project work has been performed within the s-lab – Software Quality Lab of the University of Paderborn. We will briefly describe the publications that were created in the course of this PhD thesis and explain their influence on the solution (see Figure 1.3).

	MESP Solution	Software Migration Methods	No Particular Approach
<i>Method Engineering Approach</i>	[FE16]	[GF15a] [GFS16]	[Faz+13]
<i>Method Content Definition</i>	[FLE13] [FSH14] [FSH15]	[Gri+16]	[FGS15] [GF15b]
<i>Method Tailoring</i>	[FK16] [FCE14]		[FR15] [HMF13]
<i>Method Enactment</i>	[FCE14]	[Gri+14]	[Eng+15]

Fig. 1.3 Overview of previously published material

First of all, a number of publications is directly related to our solution. An overview over our MESP framework is published as a book chapter [FE16]. This publication provides a running example that covers all tasks of the different roles defined in MESP. In another publication, we focus on the discussion of the benefits of method patterns and method services for situational method engineering [FLE13]. In particular, we show how method patterns and method services of agile and plan-driven origin can be combined. In [FK16], we describe the quality analysis framework of our approach and discuss performance evaluation results. Another publication focuses on the enactment support for method models composed with the MESP framework [FCE14]. Beside publications that directly address the framework, more practical papers describe, how such a framework could be integrated within the software development of a company. In [FSH14; FSH15], we describe in the context of a concrete method improvement initiative of an industrial partner,

how to transition from a fixed method to a method management framework that allows creating situational methods. Here, we describe how to derive method building blocks based on the existing methods, experiences, and knowledge within that company.

Beside the work on the generic MESP framework, we have also worked on a method engineering approach specific to the creation of software migration methods. Here, we described the framework for such an approach in [GF15a] and [Gri+16], the method base including fragments and patterns in [GFS16], and specific method engineering activities to improve the method definition based on experience and feedback from the method enactment in [Gri+14].

Beside the work on method engineering frameworks, we published a number of papers that reflect the experience in concrete, industrial method engineering projects performed within the s-lab – Software Quality Lab. In [Faz+13], we describe how to systematically create situation-specific requirements engineering methods and how to derive proper tool support for them. A number of publications focuses on the creation of situation-specific methods for global software development (GSD) projects. In [FSH14; FSH15], we describe an industrial project, where method building blocks were derived based on the existing practices within that company. In another publication, we explain how we improved the software engineering method and its tool support in order to improve the synchronization of team members in a GSD project [FGS15]. This is also the main theme for [GF15b] that highlights the synchronization of multiple teams in an agile GSD setting. In a further publication, we discuss how to arrive on a software engineering method in a collaborative, team-based setting [FR15].

Last, but not least, the work for this PhD thesis is also influenced by the discussions within the German-speaking community within the GI Special Interest Group on Process Models for the Development of Business Application Systems (GI-Fachgruppe Vorgehensmodelle für die betriebliche Anwendungsentwicklung). The author of this thesis is the vice chairman of the executive committee and volume co-editor of two proceedings [HMF13; Eng+15] of the annual conference.

1.4 Structure of the Thesis

The remainder of this thesis is organized as follows (cf. Figure 1.2): In Chapter 2, we introduce the background for software engineering method management, we discuss the solution requirements, and present a general overview of the related work in method engineering.

In Chapter 3, we present an overview of our solution and introduce the different roles of the MESP approach and their tasks. We go into details for these roles in the following chapters. In Chapter 4, we describe the tasks of the senior method engineer and explain the formalization of method building blocks. Chapter 5 is

concerned with the tasks of the project method engineer. Here, we go into the details of the characterization of a situation, the composition of method models, their quality assurance, and the preparation of method models for execution. In Chapter 6, we explain the tasks of the project team. Here, we explain how the method is enactment with support of the BPEL engine and how the team reflects on the method enactment to support improving method content.

We discuss the proof of concept implementation of our solution in Chapter 7. Here, we describe our prototypical tool implementation together with a first case study and scalability analysis experiment of the method composition.

We conclude the thesis with the summary of our contributions, the discussion of the solution requirements, and an outlook on possible future work in Chapter 8.

CHAPTER 2

Background

In this chapter, we discuss the background of the work presented in this thesis.

This chapter is structured as follows. In Section 2.1, we present the foundations of this thesis and clarify the used terminology. Thereafter, we discuss the requirements for our solution and the evaluation of the state of the art in software engineering management in Section 2.2. Finally, we conclude this chapter with a summary in Section 2.3.

2.1 Foundations and Terminology

2.1.1 Application Development with Software Engineering Methods

2.1.2 Situational Method Engineering

2.1.3 Executable Process Description Languages

2.2 Solution Requirements and State of the Art

2.2.1 Requirements for a Holistic Solution

2.2.2 Evaluation of Existing Approaches

2.3 Summary

2.1 Foundations and Terminology

In this section, we give a general introduction to the use of software engineering methods in application development. Thereafter, we give an overview of the field of situational method engineering that deals with the creation of custom methods for specific projects and organizations. Finally, we will briefly explain the role of executable process description languages and their relationship to the enactment of methods. Throughout the chapter, we will present the definition of key terms for this thesis.

2.1.1 Application Development with Software Engineering Methods

Software engineering is an engineering discipline. Like in other engineering disciplines, engineers apply appropriate theories, methods, and tools to find solutions. Doing that, they recognize the existing organizational, schedule-related, and financial constraints.

Definition 1 (Software Engineering). *“The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering of software.” [IEE90]*

Software engineering thereby is not limited to the technical aspects of software development. It also includes, e.g., activities of software project management or the creation of tools and theories to support software development [Som11].

The systematic and structured approach used in software engineering is called a *software development process*.

Definition 2 (Software Development Process). *A software development process is a goal-oriented activity in the context of engineering-style software development. It can be refined by subprocesses, each of which can also be refined and it usually transforms one or more input work products into one or more output work products by consuming further work products [Mün+12].*

As described, software development processes comprise project management aspects, organizational aspects, and method aspects. In this thesis, we focus on the method aspect of software development processes and use the term *software engineering method* for descriptions of the method aspect, in short, *method*.

Definition 3 (Software Engineering Method). *A software engineering method is the structured approach to create a software system. It consists of the activities, their order (process), the work products (artifacts) that are produced and used in them, and the roles that are responsible for them, as well as their description, supporting materials like tools to be used or checklists (guidances [OMGo8]), as well as the relationship between all these elements (cf. [ES10] and [Hen+14]).*

The constituents of software engineering methods are explained in more detail in chapters 4 and 5. In Figure 2.1, we illustrate further concepts that are related to methods. In the following, we describe them and their relationship.

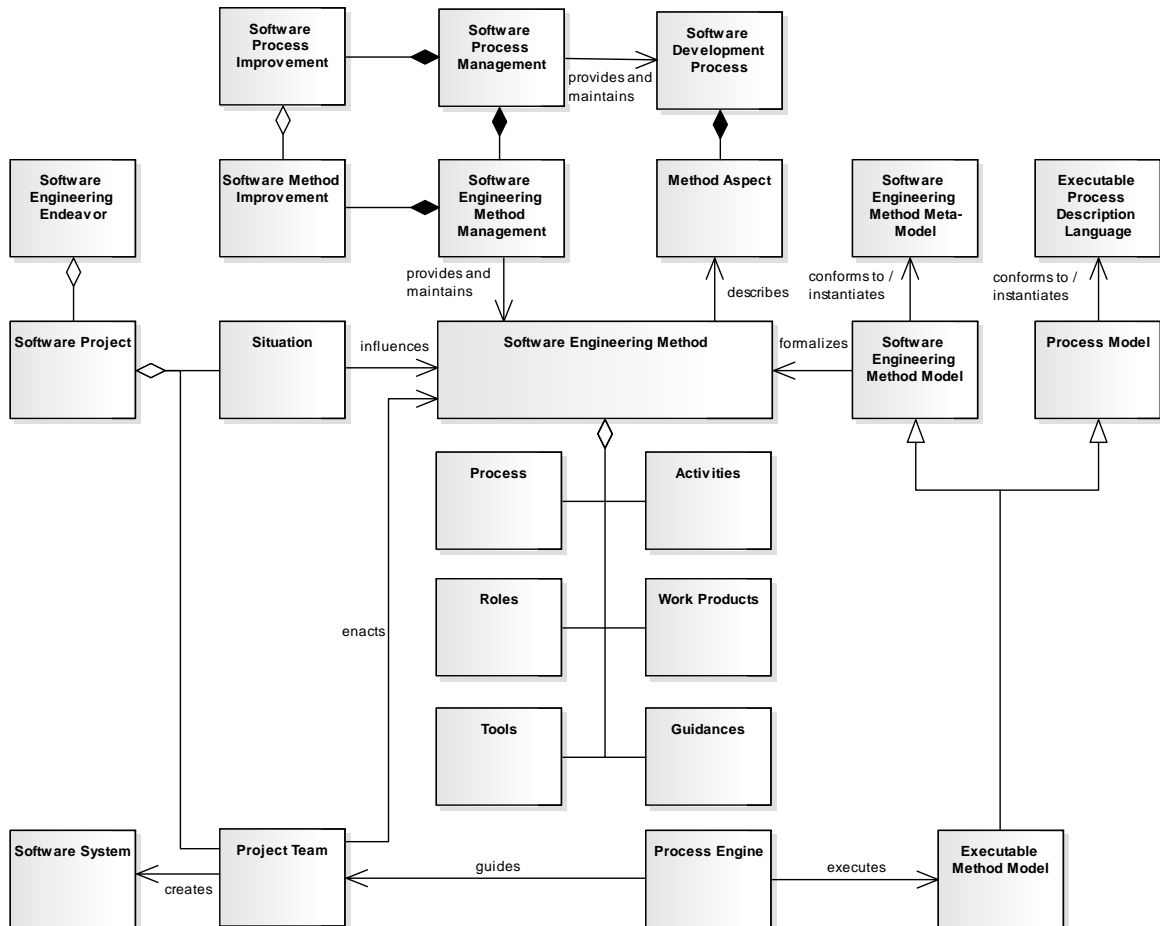


Fig. 2.1 The concepts related to software engineering methods

Software engineering method cover the typical (normally overlapping) lifecycle phases of a software project, which are software specification, software design and implementation, software validation, and software evolution [Som11]. They are enacted by a project team of a software project (or another software engineering endeavor) to create the software system.

Definition 4 (Project Team). *The project team is the group of people that performs the activities of the method (the group that enacts the method). Individual team members are assigned to the roles used in the method and thereby are responsible for performing the associated activities (adapted from [Mün+12]).*

Definition 5 (Enactment). *We speak of enactment (of the method), if the project team performs activities that are part of the method (adapted from [Mün+12]).*

Definition 6 (Activity). *An activity represents a unit of work in a method that needs to be performed in order to create the software system by following that particular method (adapted from [OMGo8]).*

Beside the term activity, we use the term *task*, when we want to stress the way *how* work needs to be performed. Activity and task are differentiated in more detail in the content chapters 4 and 5.

Definition 7 (Software Engineering Endeavor). *A software engineering endeavor is the setting, in which the method is applied, e.g., an individual software project or a whole organization (adopted from [Hen+14]).*

In case of organization-wide software engineering methods, the term *reference process* is often used instead of *method*, because it typically serves as a coarse-grained reference that needs further refinement before it can be applied. In this thesis, we focus on software projects as the software engineering endeavor.

Definition 8 (Software Project). *“A software project is a unique endeavor, which is limited by a start date and an end date and should achieve a goal.” [Mün+12]*

The use of software engineering methods benefits a software project in multiple ways (adapted from [CKO92]): first, it facilitates the understanding and communication about the activities to create a software system. Thus, it supports the coordination of those activities. Second, it offers guidance to individuals on how to carry out a specific task. Third, software engineering methods are typically the foundation for measurement and systematic improvement of the software engineering within an organization.

In order to specify methods more formally, e.g., to reduce ambiguity, they can be described as an instance of a *software engineering method meta-model*.

Definition 9 (Software Engineering Method Meta-Model). *A software engineering method meta-model is a meta-model for models that represent methods. It defines the valid structural properties of such models and can be used to instantiate conforming models. Thus, it is a specific kind of modeling language.*

We will use the term *software engineering method model* or, in short, *method model*, when we refer to the instance of such a method meta-model.

Definition 10 (Software Engineering Method Model). *A software engineering method model is a formal model of a method that is based on and conforming to a meta-model (or other formalism).*

Despite the existence of the internationally standardized meta-models Software Engineering Metamodel for Development Methodologies (SEMDM) [ISO07] and SPEM [OMGo8], very often software engineering methods are specified without using a common notation and with natural language only, e.g., [JBR99] or [SS13].

When the control and data flow within a software engineering method model is described formally enough, that is, in terms of an *executable process description language*, it can be *executed* with a *process engine*. We call this an *executable method model*.

Definition 11 (Executable Process Description Language). *An executable process description language is a language to create models that represent processes and that can be executed with a process engine.*

Definition 12 (Executable Process Model). *An executable process model is a model that is created with an executable process description language and can be interpreted by process engines (adopted from [Mün+12]).*

Definition 13 (Executable Method Model). *An executable method model is a software engineering method model that is also an executable process model.*

Definition 14 (Execution). *We speak of execution of a method model (or process model), when the method model is interpreted by a process engine, typically to support a project team in the enactment of the method model (adopted from [Mün+12]).*

Definition 15 (Process Engine). *A process engine is a generic software that can interpret the process structure and work allocation rules of a process model (adopted from [AHO2]).*

A process engine executes a process model and guides the project team. It coordinates its activities by assigning tasks to team members and by showing them guidances for their tasks. While not common for software engineering methods, business processes are commonly formalized using executable process description languages.

What software engineering method is used depends on the context of the software project, we call it the *situation*. As there is no universal software engineering method, many different software engineering methods have evolved over the past 50 years.

Definition 16 (Situation). *The situation is the sum of the local conditions of a software engineering endeavor that influence the suitability of a method or parts of it, e.g., the release frequency or the customer involvement (adopted from [Hen+14]).*

One very significant local condition (called *situational factor*) that influences as part of the situation what software engineering methods are used is that of the *application type* of the software system that is to be created. Application types are categories based on the nature of a software system (adapted from [Som11, pp. 10–11]): for example, *interactive transaction-based applications* are applications that are executed on a remote server or in the cloud and accessed by different users by their laptops, tablets, and mobile phones. They often have a large centralized database that is accessed with transaction-based requests. Examples for such

systems are modern business systems with cloud-based services and web-based or special-purpose clients (apps). Interactive transaction-based applications are often also referred to as *information systems* (e.g. in [CC05]). Another application type are *embedded systems*, which are low-level software systems that manage and control hardware devices. These embedded systems often run in resource-constrained environments and are sometimes safety-critical, for example, software that controls anti-lock braking in a car, and software in a microwave oven to control the cooking process.

In this thesis, we focus on software engineering method management for interactive transaction-based applications as our work is mostly based on software engineering for this application type. Our solution might be usable or adapted also for other types, however, an in-depth investigation remains possible future work.

Software engineering methods for interactive transaction-based applications are sometimes categorized as either *plan-driven* or *agile* [BT03]. Plan-driven refers to methods that typically differentiate multiple phases within a software project and where the activities are planned in advance and progress is measured against this plan, e.g., Quasar Enterprise [Eng08], V-Modell XT [KTF11], RUP [Kru99], or Unified Process [JBR99]. Agile refers to more lightweight, team-centered methods that typically do not define separate phases, but integrate all activities into a development cycle that is performed throughout the project, e.g., extreme programming (XP) [Bec00] or Scrum [SS13]. In agile methods, planning is incremental and changing customer requirements can be accommodated easier at the cost of continuous rework. As discussed in [BT03], both approaches have their individual advantages and drawbacks and based on the project situation, a balance has to be found.

Balancing plan-driven and agile principles based on the project context is one example for the need of *software engineering method management*. Another example is to update the method to take advantage of latest best practices [Som11]. In this thesis, we use the term software engineering method management for the activities illustrated in Figure 1.1. It includes, first, updating the defined method content, e.g., with new activities based on new best practices, second, providing methods suitable for a given situation, e.g., by balancing plan-driven and agile principles, and third, controlling the enactment of methods, e.g., by ensuring that the order of activities corresponds to the method.

The management of software engineering methods includes *software method improvement*, a term that describes the effort to continuously assess and update the method content. Closely related is the term of *software process improvement* that is used to describe the continuous assessment and change of software development processes in the whole [HRT04]. In software process improvement so-called *capability maturity models* are used as an approach to measure the degree of formality and optimization of a software development process. Two well-known capability

maturity models are Software Process Improvement and Capability Determination (formally Software Process Improvement and Capability Evaluation – SPICE and now part of ISO/IEC 15504) [Loo07] and Capability Maturity Model Integration (CMMI) [Sof10]. Capability maturity models can be seen as a generic frame of reference to assess different software engineering methods as they define requirements that must be met to reach specific maturity levels in separate disciplines.

In the next section, we will give an overview of the research field that deals with the systematic creation of software engineering methods.

2.1.2 Situational Method Engineering

Over the past 50 years, many commercial or brand-named software development methods have evolved. However, as there is no single method that is suitable for all systems, organizations, or projects, in practice, these are not widely used and especially not in their entirety (cf. [FRO03] and [KF15]). With the lack of suitable *fixed methods*, customized methods were derived by *tailoring*, i.e., modifying existing methods. However, this tailoring was often performed in an unsystematic and especially informal manner [Ped+07] with outcome of varying quality [XR08]. We refer to this unsystematic and informal approach as *free tailoring* in this thesis.

The observation that there is no single one-size-fits-all method and the need to systematically derive methods gave rise to the research area and engineering discipline of *situational method engineering* (SME) [HR10]. Situational method engineering includes all aspects of systematically creating and adapting software engineering methods based on local conditions (*situations*) as opposed to using unaltered off-the-shelf, fixed methods or unsystematic, free tailoring. SME is sometimes referred to as *method engineering* (ME), however, in strict terms SME is a subset of ME, the latter describing the systematic definition, adaption and enactment of methods for software development in general [BLW96]. Situational method engineering can be therefore seen as an attempt to provide generic solutions for software engineering method management.

Definition 17 (Situational Method Engineering). *Situational method engineering is both the research area and engineering discipline that deals with the systematic creation and adaption of software engineering methods to situations (adopted from [HR10]).*

Situational method engineering approaches can be compared among several dimensions. Looking at the two dimensions *degree of flexibility* and *tailoring effort*, method engineering approaches can be grouped roughly into three groups:

- the group of creation-based,
- the group of assembly-based, and
- the group of configuration-based approaches.

As illustrated in Figure 2.2, each group represents its individual trade-off between the two dimensions (cf. [HBJ94] and [Hen+14]): going from left to right, the flexibility of the approach in terms of adaption to a specific situation increases, however, the effort to create an unflawed method is also higher, thus rendering some approaches to cumbersome for software engineering method management. Situational method engineering covers the area between two extremes: On the one hand, using fixed methods without any adaption and on the other hand, free tailoring with potentially changing anything without following any underlying consistency rules. In the following, we characterize the three groups within situational method engineering.

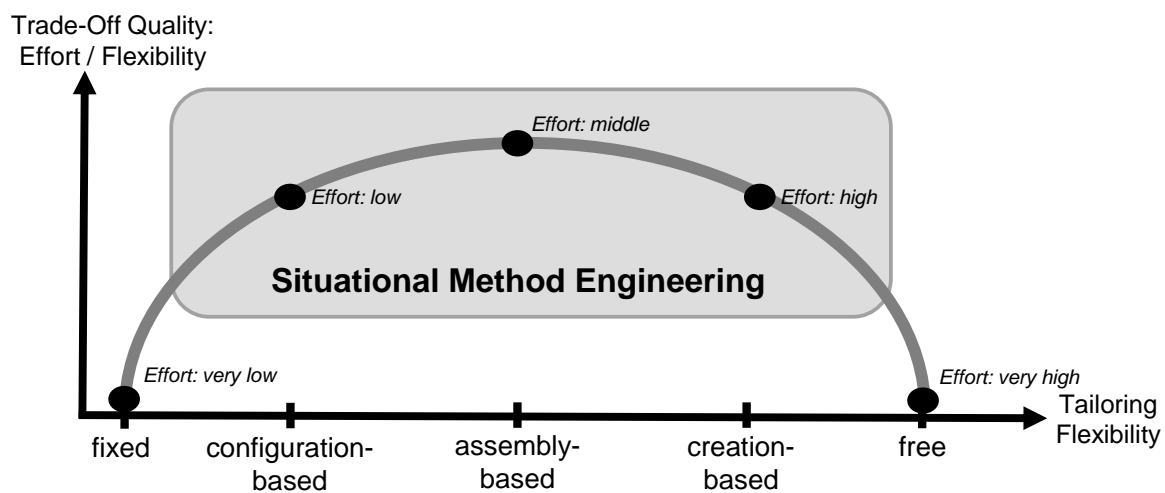


Fig. 2.2 Trade-off between effort and flexibility of different groups of method engineering approaches

Creation-based Approaches

In creation-based approaches, a method is created from scratch, however, its constituents are instantiated from a common meta-model. This meta-model defines the available concepts and their relations to describe a method. Therefore, the use of meta-models allows ensuring a basic level of formality. In addition, it provides a basic level of disambiguation as two methods can be compared with respect to the instantiated meta-classes. Consequently, meta-models are often used as a formal foundation for (potentially fixed) methods and are not primarily used to facilitate situational method engineering.

Meta-models of software engineering methods that aim at being used for situational method engineering include SEMDM and SPEM. Beside these well-known standards, other meta-models have been proposed, either as stand-alone (e.g., MetaMe [ES10]) or as an extension to these standards (e.g., eSPEM [Ell+10]).

Regarding the two dimensions of situational method engineering explained before, creation-based approaches provide a lot of flexibility at the cost of considerable tailoring effort. With respect to the software engineering method management layers (cf. Figure 1.1), methods can be defined and tailored to a project from scratch with the most possible freedom only constrained by the concepts of the meta-model. However, this freedom requires a lot of method engineering expertise to tailor meaningful methods. In addition, as reuse is not particularly addressed, considerable effort is necessary to define the contents of a complete method. Apart from academic feasibility studies, this approach is therefore rather inapplicable, particularly for method engineering in the context of complex software engineering endeavors. For example, researchers in a recent study complain that they could not find a single publication for a method based on SEMDM [KFS13a].

Assembly-based Approaches

In assembly-based approaches, component-like concepts to enable modularity and reuse play an important role. Instead of creating a method from scratch, existing method building blocks are reused. The basic idea is to maintain a repository of predefined method building blocks. Based on this method repository, a method is then created by assembling method building blocks suitable for the project. Since building blocks are also instantiated from a meta-model, assembly-based method engineering is based on creation-based method engineering.

As there is no common standard, approaches follow their own component model and terminology. In many approaches, building blocks represent either product or process aspects and are then often referred to as *method fragments* (e.g., [HB95], [Bri96], [Cer+11]) or the combine both and are then mostly termed *method chunks* (e.g., [SRG96; Pli96],[RP96b],[Ralo4]). In [KLR96] building blocks are named components instead of fragments. In recent publications the notion of *method services* and *method-as-a-service* are discussed [Rolo9].

In general, assembly-based approaches have the advantage that they address reuse of method content explicitly, which reduces the effort to tailor a method (cf. Figure 1.1). The flexibility is limited mainly by the available method building blocks that were defined. These, however, can be added on the fly. This makes assembly-based approaches more flexible than configuration-based ones. Nevertheless, despite the balance between flexibility and effort that assembly-based approaches provide, they still have limited relevance in practice. Only few examples of practical application are documented, e.g., [HS05] and [Wee+06]. We believe, this is partly due to the limitations of contemporary approaches, discussed in Section 2.2.2, especially the lack of high level-modeling languages to define and compose method building blocks with suitable granularity and the lack of sufficient tool support.

Assembly-based method engineering needs to comprise the following tasks (cf. [BLW96]) on the three conceptual layers of software engineering method management (see 1.1) shown in Figure 2.3. On the layer of method content definition: *Extract Reusable Method Content* and *Define Method Building Blocks*. On the layer of method tailoring: *Characterize Project*, *Compose Project-Specific Method*, and *Assure Quality of Method*. On the layer of method enactment: *Coordinate Activities* and *Guide Tasks*, as well as *Reflect Method*.

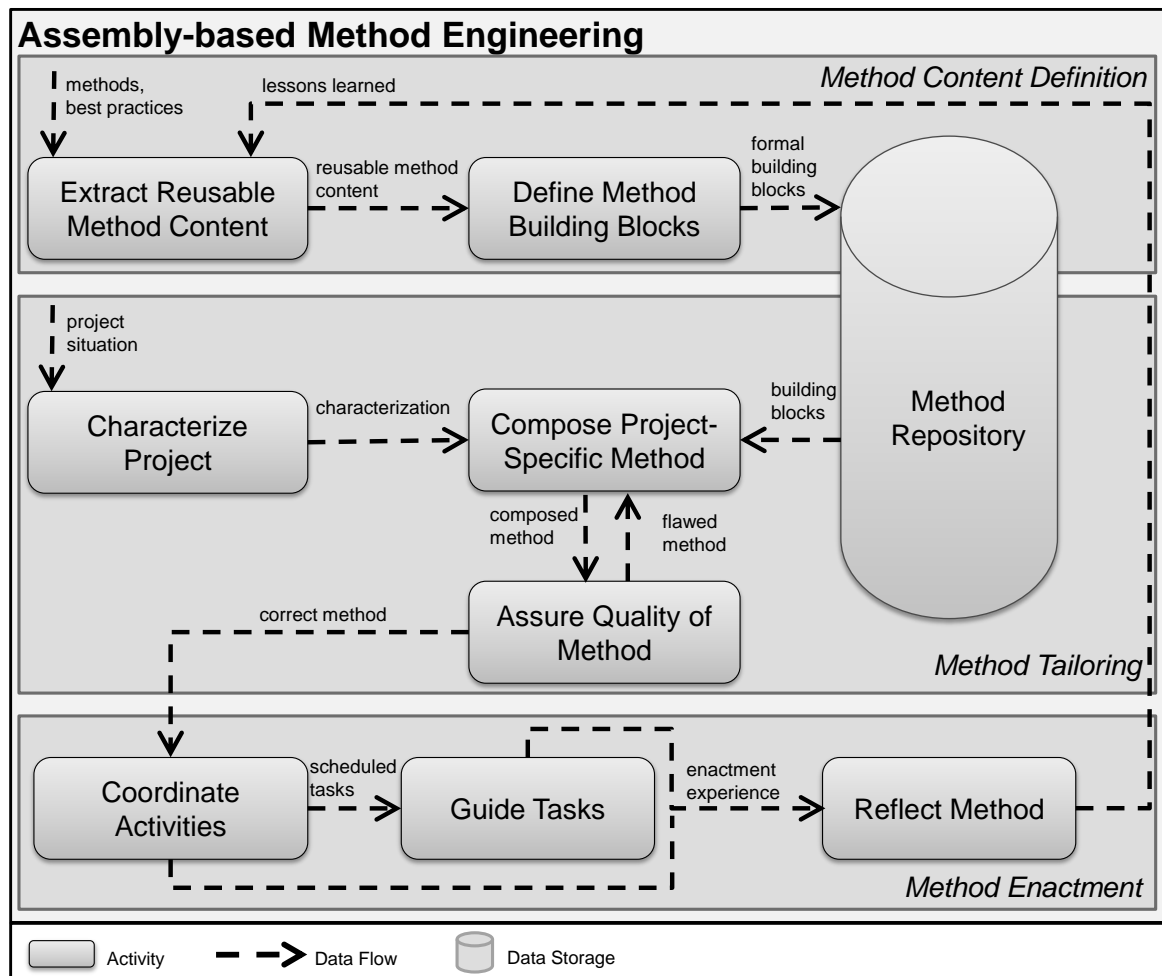


Fig. 2.3 Tasks of assembly-based method engineering

Extract Reusable Method Content is the task that needs to be performed to identify method knowledge that shall be made available for incorporation into situational methods. Possible sources for method knowledge are, for example, published methods or established practices used within the organization [Ralo4],[FCE14]. With *Define Method Building Blocks* the extracted information is formalized and stored into the method repository. All building blocks are characterized in terms of suitability for different project situations.

Based on the building blocks defined in the method repository, a situational method can be composed. Firstly, with *Characterize Project* the situation of the planned development project is assessed, such that, secondly, suitable building blocks can be identified during *Compose Project-Specific Method*. Here, the actual method is composed according to the needs of the situation. With *Assure Quality of Method* it is ensured that the method contains no flaws such that it can be enacted later on.

The task *Coordinate Activities* helps the project team to identify and assign the pending tasks and work products to use, while the task *Guide Tasks* helps them to carry out the software engineering tasks according to the specified method.

Configuration-based Approaches

In configuration-based approaches, a method is created by selecting a value from a range of choices for each provided configuration point. This set of choices, the *configuration*, is then used to derive a method automatically. As the solution space of possible methods is limited (and in comparison not that big), it is feasible to ensure the consistency of all possibly configurable methods (i.e. removing inconsistent configurations) upfront. Thus, normally, methods of configuration-based approaches are consistent by design and do not require additional quality assurance by the user of the approach. However, this also means that additional method content cannot be added by the user. Since the so-called *base-method* and its configuration choices are typically formalized, using a meta-model, configuration-based method engineering is based on creation-based method engineering.

Examples of configuration-based approaches are MC Sandbox [KÅ05] and V-Modell XT [HH08]. The latter is well-known on the German market as it is a requirement for government IT projects there. It focuses on the work products that need to be created and defines only coarse-grained activities that allow for further refinement by (manual) tailoring.

The strength of configuration-based approaches is that integrity and quality of the resulting method are guaranteed and do not have to be addressed explicitly during method tailoring. Therefore, these approaches require only little method engineering expertise to create a method. However, the solution space of possible methods is limited to the foreseen configurations. Consequently, the flexibility is limited. It is insufficient for complex software engineering endeavors.

2.1.3 Executable Process Description Languages

One purpose of software engineering methods is to describe the flow of activities, often termed *process*, and thereby supporting the coordination of method enactment. Based on the idea of process programs [Ost87], the idea emerged to model methods with *executable process description languages*. These languages allow describing

processes with sufficient syntax and semantics to simulate them or to execute them with a *process engine* (called *workflow management software* in [AHO2]). The process engine could ensure that the enactment of the method conforms to its specification and it could support the coordination of the activities by taking care of the control and data flow [GJM03]. For the interaction with the team members, the process engine offers a *task management component*. If an activity has to be carried out by a team member, the task management component creates a task for her and directs the results back to the executed process.

However, executability is not supported by standard method meta-models like SEMDM or SPEM [Ben+10]. Therefore, other languages have been proposed to model executable methods, either stand-alone, e.g., Little-JIL [Wis+00], or as an extension to SPEM, e.g., eSPEM [Ell+10; Ell+11]. In general, these approaches have in common that they lack situational method engineering and mature tool support in terms of a process engine that supports both control and data flow.

While the use of executable process description languages and their support by process engines is an emerging topic for the domain of software engineering methods and method engineering, it is quite common in the domain of business processes. Here, languages like Event-driven Process Chains (EPC) [KNS92], Business Process Execution Language (BPEL) [OAS07] with its extension BPEL4People [OAS10], and Business Process Model and Notation (BPMN) [OMG11] are used to model business processes and to execute them with process engines. Especially, the *business process modeling languages* BPEL and BPMN are de facto standards for modeling business processes that potentially span different organizations and have a long life span. There exists a wide range of mature tools to support the creation and execution of these *business process models*. Generally, the focus of the business process domain is to derive technical implementations (software systems) from business processes sometimes called *business-driven development* (BDD) [Mit05]. While human interaction is important, the ultimate goal is to specify the coordination of automated services. The coordination of humans and especially supporting the established concepts and terminology of software engineering methods is not supported by standard business process modeling languages and their process engines.

2.2 Solution Requirements and State of the Art

In this section, we discuss the solution requirements of a holistic software engineering method management solution. Thereafter, we present the state of the art and assess existing approaches with respect to the solution requirements. While we discuss global related work in this chapter, we introduce more specific, topic-related works later in the respective chapters.

2.2.1 Requirements for a Holistic Solution

In this section, we revisit the challenges of software engineering method management introduced in the introduction of this thesis. First, we discuss the suitability of the situational method engineering approaches presented in Section 2.1.2 with respect to these challenges. Then we define solution requirements (SRs) for a software engineering method management solution addressing these challenges. We use these requirements later to evaluate the related work and to assess our proposed solution.

Necessity of an Assembly-based Approach

In Section 1.1, we discussed the challenges for software engineering method management on each layer of the hierarchy:

Challenge 1 – Method Content Definition To enable updates of method content based on new trends, best practices, and lessons learned

Challenge 2 – Method Tailoring To enable the creation of consistent software engineering methods for specific situations based on defined method content

Challenge 3 – Method Enactment To enable the proper enactment of the tailored method according to its definition

In Section 2.1.2, we discussed different groups of situational method engineering approaches. In order to address Challenge 1, a solution needs to be flexible enough to update the method content continuously. As the available method content is unchangeable in fixed and configuration-based approaches, these cannot address Challenge 1 sufficiently. In order to address Challenge 2, a solution needs to offer the definition and reuse of method content. As method content is not defined to be reused later, this is not the case for creation-based approaches and free tailoring. Thus, a solution for software engineering method management needs to be assembly-based. First, it needs to allow the creation of additional method building blocks to reflect new trends, best practices, and lessons learned. Second, it needs to allow for the creation of a method model based on the composition of suitable method building blocks with respect to a project situation. Third, it needs to support the enactment of the composed method models.

Refined Solution Requirements

In [Ell+11], Ellner et al. describe common requirements for software engineering method management solutions based on previous work of other authors [CJ99],[JBD99],[Gru02]. We present their common requirements in Table 2.1.

Table 2.1 Common requirements for software engineering method management solutions without refinement regarding software engineering management layers or assembly-based method engineering (adapted from [Ell+11])

<i>Common Requirement</i>	Description
<i>Scalability</i>	Large as well as small methods can be created and managed
<i>Decomposability</i>	Sub methods and compound methods can be defined
<i>Adaptability</i>	Tailoring a given method to the needs of a project must be straightforward
<i>Testability</i>	Plausibility checks can be performed automatically to help creating methods
<i>Easy-to-digest formalism</i>	Needed, because in the past complex formalisms have prevented adoption by practitioners
<i>Executability</i>	Methods can be directly interpreted by a machine or otherwise mapped to an executable language
<i>Automatic process execution</i>	A process engine is provided that supports the team in their work according to the method, triggers certain tasks on time, and controls the delivery of artifacts
<i>Electronic process guide</i>	A process engine actively guides team members by providing information that is sensitive to the task context at hand according to the method
<i>Automatic audit trail</i>	A process engine automatically keeps track of changes to work products and progress of the method enactment

While the list of common requirements provides a good starting point, it can be improved in two ways. First, the requirements are not explicitly associated with the method engineering management layers or the described challenges. In particular, requirements for method content definition and method tailoring are not clearly separated. Thus, their relationship to the challenges is not stated explicitly. Second, the requirements are very generic and do not take into account specifics of assembly-based approaches.

Based on the common requirements and characteristics of assembly-based approaches we refined SRs for a solution that addresses the software engineering method management challenges. The relationship between software engineering method management layers, common requirements and our solution requirements is shown in Figure 2.4. In the following, we discuss our solution requirements. Following the different levels of the software engineering method management hierarchy, we first discuss five method content-related criteria, afterward four tailoring-related criteria, and finally, three enactment-related criteria.

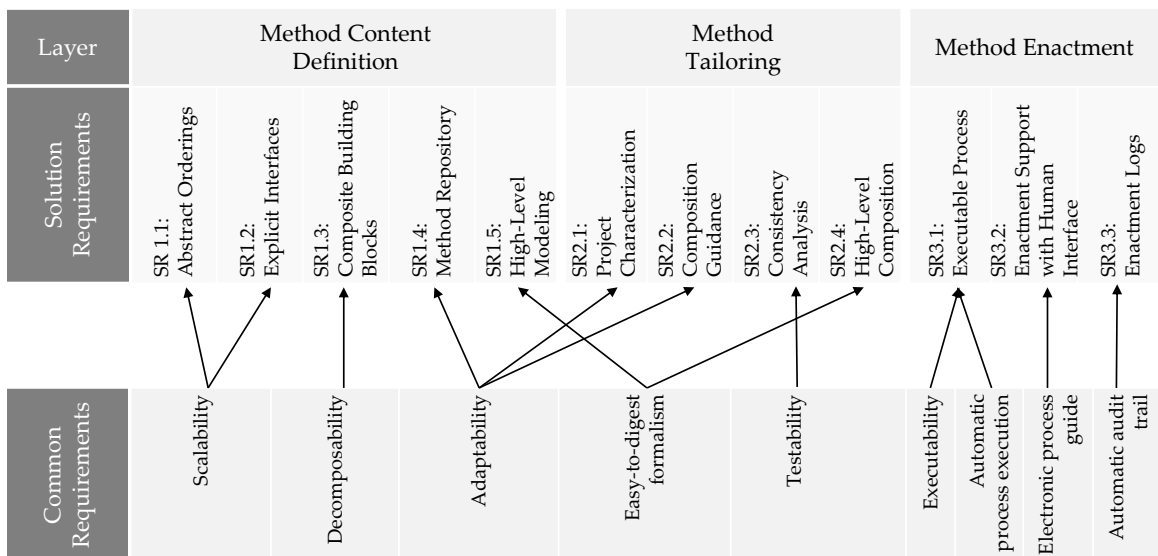


Fig. 2.4 The relationship between software engineering method management layers, our solution requirements (SRs), and the common requirements in [Eil+11]

Solution Requirement 1.1 – Abstract Orderings As a refinement of the *scalability* requirement, we define the criterion *abstract orderings*. Abstract orderings allow modeling orderings without referencing concrete building blocks. This ensures that the orderings can be used also for yet not defined building blocks. These ordering can then be used to speed up the definition of especially large methods and they serve as an additional guidance during method composition.

Solution Requirement 1.2 – Explicit Interfaces As another criterion that is related to the *scalability* requirement, we define the criterion *explicit interfaces*. Explicit interfaces abstract from the content of method building blocks and make the information that is necessary to find and compose suitable building blocks available. Thus, they reduce the cognitive load when performing the related tasks.

Solution Requirement 1.3 – Composite Building Blocks As a refinement of the *decomposability* requirement, we define the criterion *composite building blocks* that assesses whether building blocks of an approach support the composite pattern. Composite building blocks allow hiding the complexity of building block compositions as they can be treated like ordinary building blocks.

Solution Requirement 1.4 – Method Repository As a refinement of the *adaptability* requirement, an assembly-based approach requires a method repository to store method building blocks. Some method engineering approaches additionally store composed methods and method enactment data in the method repository.

Solution Requirement 1.5 – High-level Modeling As a refinement of the *easy-to-digest formalism* requirement, this criterion expresses whether an approach offers a high-level modeling language and appropriate tooling to define method building blocks. This would ease the definition of method building blocks.

Solution Requirement 2.1 – Project Characterization As a refinement of the *adaptability* requirement, an assembly-based approach requires the characterization of the project in order to ease finding suitable building blocks for a project.

Solution Requirement 2.2 – Composition Guidance As another refinement of the *adaptability* requirement, this criterion expresses whether an approach offers guidance in composing a suitable method for a project.

Solution Requirement 2.3 – Consistency Analysis As a refinement of the *testability* requirement, we define the criterion consistency analysis. It measures, whether an approach offers consistency rules and tooling for it. This criterion is partially fulfilled, if the conformance to the meta-model can be checked automatically. It is fulfilled, if additional plausibility and quality checks are performed.

Solution Requirement 2.4 – High-level Composition As a refinement of the *easy-to-digest formalism* requirement, this criterion expresses whether an approach offers a high-level composition language and appropriate tooling to define method compositions. This would ease the composition of methods.

Solution Requirement 3.1 – Executable Process As a refinement of both, the *executability* and the *automatic process enactment* requirement, this criterion measures whether methods can be executed with a process engine. This criterion is fully fulfilled, if both flow of activities and the flow of work products is considered and supported.

Solution Requirement 3.2 – Enactment support with Human Interface As a refinement of the *electronic process guide* requirement, this criterion expresses whether an approach offers guidance for the enactment of a task and the GUI for the team member to interact with the method. This criterion is fully fulfilled, if the flow of work products is considered. This means that the team member can inquire about her input work products and report output work products.

Solution Requirement 3.3 – Enactment Logs As a refinement of the *automatic audit trail* requirement, this criterion expresses whether an approach offers support to log information about the enactment of a method in order to support the reflection of the method.

The solution requirements are used to assess approaches proposed in related work in the following section. Later, we use the solution requirements to assess our proposed solution.

2.2.2 Evaluation of Existing Approaches

In this section, we discuss existing holistic tool-supported approaches for assembly-based method engineering and assess them against our solution requirements.

Overview of Existing Approaches

Related works for assembly-based method engineering exist on a broad range, however, often focusing on specific aspects. In the following, we give a brief overview of existing works and thereby focus on holistic solutions to assembly-based method engineering that are tool-supported. These works will be considered for the evaluation later in this chapter, while additional works will be also discussed in this section for the purpose of completeness. References to related work concerning specific aspects of assembly-based method engineering, such as modeling of executable processes, are not considered here but later in the respective chapters.

Holistic Assembly-based Approaches In the following, we give a brief overview of works that provide holistic solutions to assembly-based method engineering and that are tool-supported. These works will be considered for the evaluation later in this chapter, while other works will be also discussed here for the purpose of

completeness. Brinkkemper and Harmsen presented an assembly-based method engineering approach that is accompanied by their tool *Demacrone* [Har97]. Their approach covers the definition of method building blocks and their storage in a method repository [HB95]. They also define dimensions for the project characterization and formulate assembly rules for the composition of methods that could be reused for later consistency analysis. Their solution offers the generation of a CAME environment and a process engine based on the composed method. However, they admit issues with the usability, performance, and reliability of their approach [Har97, p. 277]. In addition, their solution depends on low-level modeling languages and on outdated and proprietary technologies as the Prolan programming language of the CASE environment Maestro II and a character-based user interface.

The method engineering approach for the tool *MERU* [GP01] is based on the creation of a implementation-independent Method Requirements Specification (MRS) by means of a MRS Creator. This MRS is used instead of the characterization of the project. The MRS can be analyzed for basic consistency violations and is then translated semi-automatically in several steps to method building blocks and then a composed method. From this method a suitable CASE environment is generated afterward [PS97]. The authors do not explain whether created method building blocks are supposed to be used for future methods. In addition, support for process enactment is missing.

Another tool-supported approach is *MENTOR* [SRG96; Pli96]. This approach uses method building blocks that follow the NATURE contextual approach [Gro+97]. Thus, building blocks are tree-based structures that bundle *situations* to *decisions*, where situations reflect the state of products and decisions the intention of the engineer. Instead of characterizing a project and modeling the whole process explicitly, the core of the solution is the Guidance Engine that finds suitable building blocks according to a situation and the defined high-level process. Based on the selection of the method engineer, it executes them and thereby offers guidance to the application developer (or method engineer). The solution includes a method repository, but there is no support for consistency analysis and like *Demacrone* the tooling is based on outdated technologies.

The tool family *Method Management Tools* [KLR96] is build on top of the meta-CASE environment MetaEdit+, whose successor is still available for current software platforms. It offers a method repository to store method building blocks, means to compose methods, basic means to check methods for consistency, and generators to generate a CASE tool customized to the method. Not covered by the solution is supporting the choice of suitable method building blocks and the enactment of the method with the guidance of the project team.

An assembly-based approach that was proposed recently and that is based on modern technologies and standards is *MOSKitt4ME* [Cer+11]. It is build on top of an Eclipse-based [Ste+09] modeling platform called MOSKitt. The approach is

based on the SPEM standard and supports the composition of methods based on a method repository of method building blocks and specifically focuses on deriving a suitable CASE environment for the method. Therefore, technical building blocks are part of the method definition. These define the tools that should be used to create and to process work products. After method composition these tools are automatically bundled into a CASE environment to use during method enactment. The approach has no process execution support as execution semantics are missing in SPEM. A consistency analysis and also support for abstract orderings is also missing.

Ellner et al. [Ell+10; Ell+11] provide *eSPEM*, a tool-based approach for the modeling and execution of methods. The approach is based on extensions to SPEM to address SPEMs lack of executability and it is based on the Eclipse ecosystem [Ste+09]. It offers a method repository for the storage of method building blocks and visual editors to model reusable method building blocks and compose them to executable methods. The approach also offers rudimentary analysis support of created methods. While the extension of SPEM with execution support is one of the strengths of this approach and its tooling, there is no explicit support for situational method engineering in terms of characterizing method building blocks and the project nor for abstract orderings.

Further Related Work Bendraou et al. [Ben+07] also propose an extensions to SPEM called xSPEM to address SPEMs lack of executability. However, creating situational methods is not in scope of their work and they only sketch some mappings between concepts of xSPEM and BPEL and, e.g., human interaction with executed methods is not discussed.

In addition to these approaches, related work comprises other approaches worth mentioning here. Software engineering method frameworks (termed “process framework” in [KFS13a]) like V-Modell XT [KTF11] or Rational Unified Process (RUP) [Kru99] offer more flexibility than rigid methods and allow a certain degree of tailoring to a project or organization. These frameworks can be attributed to configuration-based method engineering, however, the offered guidance and tool support for tailoring varies among approaches. V-Modell XT for example offers explicit configuration points that lead to the inclusion or exclusion of activities in the method. Additionally, methods of software engineering method frameworks are based on a meta-model that provides a formal foundation and allows additional creation-based method engineering.

Another configuration-based approach consists of the Method for Method Configuration and its tool support MC Sandbox [KÅ11; KÅ12]. Here, non-hierarchical *method components* are defined that express how to transform input work products into a defined output work product together with the *rationale* of such a transformation. For each assignment of a so-called *characteristic* (basically a configuration

point) a configuration package is created that describes which method components shall be included. Multiple characteristics are then covered by *configuration templates* that basically characterizes a complete development situation and a complete set of required method components. When changing the assignment for single characteristics, the tooling can compute the set of included method components and detect classification conflicts (contradicting method components) that then have to be handled manually. The enactment is supported with a project-specific method website that is derived from the included method components.

In addition to the described method engineering approaches, various works propose approaches that focus on the conceptual level. We exemplarily present three of them. A more complete overview is provided in [Hen+14]. Spijkerman [Spi15] proposes the extension MetaMe++ as an improvement to the creation-based method engineering approach described in [ES10]. Here, method requirements are derived based on an as-is analysis of the specific situational context of an existing organization. These are then used to iteratively improve the method in several method improvement iterations. The method is thereby modeled based on the product and process meta-model of MetaMe++.

The work by Geisen [Gei15] deals with the adaptation of a method during method enactment. Adaptions of the method are seen as necessary reactions to changes of the situational context of the project. The work describes a conceptual framework that is based on the adoption of the MAPE-K feedback loop [KC03] from the adaptive systems domain.

Ralyté proposes two approaches to extract reusable method building blocks from existing methods and from scratch in [Ralo4]. She uses the map formalism [RP96b] to model the approaches where a number of *intentions* is linked to a number of *strategies*.

Another related group of work are ALM suites [KV09] that integrate with various tools, e.g., configuration management systems, change management systems, and IDEs into a distributed development and collaboration platform. Examples for these suites are Microsoft Team Foundation Server⁴ (TFS), IBM Rational Team Concert⁵, and Method Park Stages⁶. These ALM suites offer to some extent method execution support and allow for the use of automated workflows defined in templates delivered by the ALM vendor. These workflows usually cover only limited parts of the whole software engineering method, e.g., they cannot determine the next possible steps in a process and guide team members accordingly [Ell+11, p. 80]. Manual adaption of these templates is possible, but especially the creation of situational methods is not supported explicitly. In [KKT14] Kuhrmann et al. describe a generic framework that allows generating templates for ALM suites like

⁴<http://www.visualstudio.com/en-us/products/tfs-overview-vs.aspx>

⁵<http://www.ibm.com/software/products/en/rtc>

⁶<http://stages.methodpark.de/>

TFS, based on process models. However, situational method engineering is also not addressed explicitly here.

Evaluation

Existing related works for assembly-based method engineering exist on a broad range, however, often focusing on specific aspects. In our evaluation, we consider the holistic, tool-supported assembly-based method engineering approaches discussed in this section. Hence, we omit approaches that are not assembly-based ([KTF11], [Kru99], [KÅ11], [KKT14]), that focus only on specific layers of the software engineering method management ([Ben+07], [Ralo4]), or that do not provide any tool support ([Spi15], [Gei15]). Table 2.2 shows the result of our evaluation.

Table 2.2 Evaluation of existing tool-supported approaches for assembly-based method engineering

	Content Definition					Tailoring Support				Enactment Support		
	abstract orderings	explicit interfaces	composite building blocks	method repository	high-level modelling	project characterization	composition guidance	consistency analysis	high-level composition	executable process	enactment support with human interface	enactment logs
Demacrone [HB95],[Har97]	-	-	□	+	-	+	(+)	-	-	(+)	-	-
MERU [GP01], [PS97]	-	-	□	□	□	(+)	(+)	(+)	□	-	-	-
MENTOR [SRG96],[Pli96]	-	+	□	+	(+)	-	(+)	(+)	-	(+)	(+)	□
Method Management Tools [KLR96]	-	-	□	+	+	-	-	(+)	+	-	-	-
MOSKitt4ME [Cer+11]	-	+	□	+	+	-	-	-	+	-	-	-
eSPEM [Eil+10], [Eil+11]	-	(+)	+	+	+	-	-	(+)	+	+	+	+

+ supported

(+) partially supported

- not supported

□ not considered

Content Definition While several approaches differentiate different levels of granularity, none of the approaches provides support for *abstract orderings*. About half of the approaches support the definition of *explicit interfaces* for method building blocks and allow modeling them with high-level modeling support. While MENTOR and Method Management Tools use outdated technologies, the approaches eSPEM and MOSKitt4ME are build on top of the contemporary Eclipse platform and use the popular Ecore meta-model [Ste+09]. Except eSPEM, none of the approaches discusses *composite building blocks*. Except of MERU, where it is unclear, all

approaches offer a *method repository*. About half of the approaches support *high-level modeling* with visual languages. As stated, eSPEM and MOSKitt4ME are build on top of the contemporary Eclipse platform, while Method Management Tools and MENTOR use outdated technologies.

Tailoring Support Regarding tailoring support, only the two approaches Demacrone and MERU offer means for *explicit project characterization*. In the latter approach, instead of the typical situational factors, a method requirements specification is created. *Composition guidance* and *consistency analysis* are provided only partially and only in roughly half of the approaches. In addition, only MERU and MENTOR provide both. Regarding *high-level composition* support, besides one exception, the support looks the same as that of the modeling of method building blocks. The exception is that MERU does not offer high-level composition support, as processes are not explicitly modeled in the approach.

Enactment Support Most of the presented approaches offer, if at all, only weak enactment support. eSPEM is the only approach that supports all enactment-related criteria, however, it has its particular weakness in method tailoring. The only other approaches that offer some enactment support are Demacrone and MERU.

2.3 Summary

In this chapter, we provided the necessary background for the remainder of the thesis. We explained software engineering methods, discussed their creation with situational method engineering, and presented an evaluation of the state of the art in assembly-based method engineering. Based on this foundation, we now present our solution for software engineering method management that specifically addresses the weaknesses of the other approaches. We start off with an overview of our solution in the following chapter.

CHAPTER 3

Solution Overview

In this chapter, we provide a general overview of our solution and an end-to-end example. Details will then be discussed in the following chapters.

This chapter is structured as follows. We provide an overview of our solution for software engineering method management in Section 3.1 and discuss MESP roles, work products, and tooling. In Section 3.2, we illustrate our solution with a running example. We conclude the chapter with a summary in Section 3.3.

3.1 Overview of the MESP Approach

3.1.1 Overview of MESP Roles

3.1.2 Overview of MESP Work Products

3.1.3 Overview of MESP Tools

3.1.4 Integrated Overview of MESP Solution

3.2 End-to-End Example

3.2.1 Method Content Definition

3.2.2 Method Tailoring

3.2.3 Method Enactment

3.3 Summary

3.1 Overview of the MESP Approach

Our evaluation presented in Section 2.2.2 revealed that existing approaches do not fulfill the requirements for software engineering method management. More pre-

cisely, there is no holistic solution that, first, enables the updates of method content based on new trends, best practices, and lessons learned, that, second, enables the creation of consistent software engineering methods for specific situations based on defined method content, and that, third, enables the proper enactment of the tailored method according to its definition.

MESP is our proposed approach that addresses the stated requirements by offering a holistic solution for software engineering method management. First, following the assembly-based idea of “define once, use many”, method content is defined in a formal, reusable way. Second, method models are consistently tailored to a specific project by reusing and combining useful method content. Finally, method models are executed with a process engine, so the enactment of the method by the project team is supported.

Our solution describes the MESP roles and tasks necessary on all three layers of the software engineering method management hierarchy. It also includes the formal models that describe all necessary elements to define and characterize method building blocks and analyzable, executable method models. We also provide tool support to create these models, to check the consistency of method models, and to execute them with standard off-the-shelf process engines. Thus, we provide a meta-method, a method to create software engineering methods, with all its required aspects as illustrated with the UML class diagram in Figure 3.1.

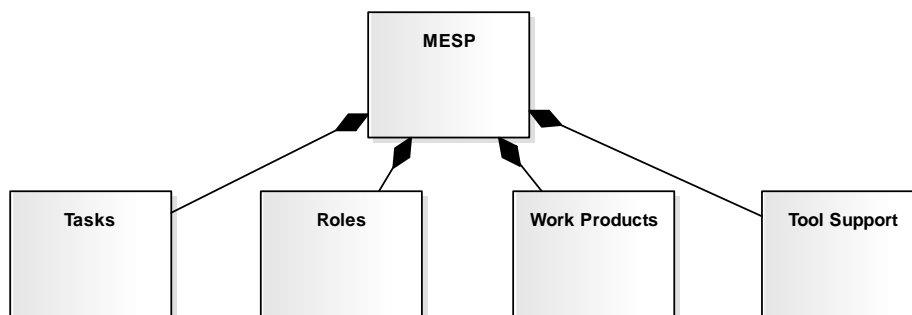


Fig. 3.1 The aspects included in our MESP solution

3.1.1 Overview of MESP Roles

In MESP, we differentiate three different roles, where each MESP role is responsible for a layer of the software engineering method management hierarchy. Figure 3.2 illustrates the three MESP roles and their tasks. In general, we differentiate between the general, continuously performed tasks of the MESP role senior method engineer to define reusable method content and the project-specific tasks of the other MESP roles for one particular project. Potentially, the method content that is defined once can be reused in many different projects. Thus, the tasks of the senior

method engineer are only loosely coupled to the tasks of the other MESP roles. In the following, we briefly describe each MESP role and its tasks.

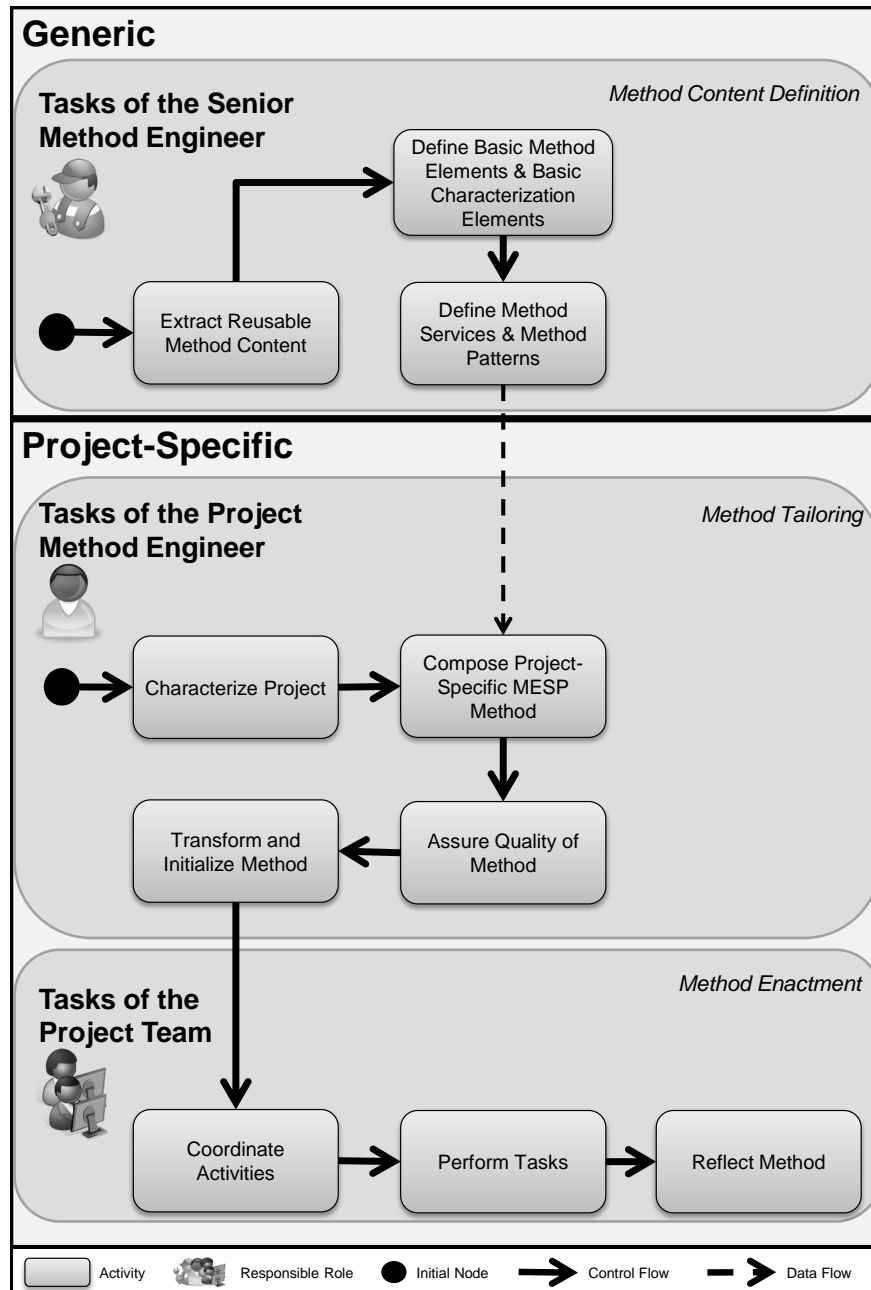


Fig. 3.2 Overview of the MESP Tasks

Senior Method Engineer Senior method engineers are responsible for defining reusable method building blocks. First, they have to *extract reusable method content*, e.g., by investigating the existing software development within an organization

as we described in [FCE14] or by extracting it from existing method descriptions in literature as described in a master thesis [Sie15]. Next, they formalize method content, and *define basic elements and method building blocks*, and store them in a method repository as we described in [FE16]. Method building blocks are based on shared basic elements that ensure consistency and interoperability between method building blocks. If suitable basic elements are missing, they are created as required. Method content can be formalized using the two types of method building blocks, method services and method patterns.

The MESP role of the senior method engineers is to be filled in with quality managers, SPI managers, or very experienced project managers with modeling experience. The tasks of the senior method engineer are exemplified in Section 3.2.1 and discussed in more detail in Chapter 4.

Project Method Engineer Project method engineers are responsible for defining a suitable method model for their respective project. Thus, they first *characterize their project* using basic elements from the method repository. Then they choose suitable method services and method patterns and *compose them to a project-specific method* model for their project as we described in [Faz+13]. They have to *assure the quality of the method* model using the automated quality assurance framework of the MESP tool support as we described in [FK16]. Thereafter, they *prepare and initialize the method* model. They prepare the method model for execution by transforming it into a process model and deploy it into a standard process engine as described in [FCE14]. In addition, they assign team members to roles used in the method model.

The MESP role of the project method engineers is to be filled in with project managers that are responsible for managing the project work, its tasks, its outputs, and its team members. The tasks of the project method engineer are exemplified in Section 3.2.2 and discussed in more detail in Chapter 5.

Project Team The project team enacts the composed method model to create the software system in their software project. They execute the transformed process model in a process engine that *coordinates their activities* and provides them with guidance on the pending tasks as we described in [FCE14]. The process engine ensures that the project team *performs its tasks* as prescribed and provides them with task descriptions. In order to support the senior method engineers in improving method services and method patterns, the project team *reflects the enactment of the method* and provides feedback to her by capturing lessons learned as we described in [Gri+14].

The MESP role of the project team is filled in with the members of the project team that carry out the tasks of the project. The tasks of the project method engineer are exemplified in Section 3.2.2 and discussed in more detail in Chapter 6.

3.1.2 Overview of MESP Work Products

Figure 3.3 illustrates the work products that are created as part of the MESP approach. They can be grouped into four groups: The first group consists of work products related to Method Building Blocks maintained by the senior method engineer. The second group consists of work products to compose a MESP Method Model for a project. The third group are the work products to express method models in the executable process description language BPEL. They are used to transform method models to BPEL Process Models in order to execute them with a standard process engine. The fourth group consists of work products that are related to the Method Enactment and execution of the model.

Method Building Blocks Regarding the first group, basic elements (Basic Method Elements and Basic Characterization Elements) are used to characterize Method Services and Method Patterns, but also method models via their respective Interfaces. While they are used for method services and method patterns to express what is provided and whether the building block is suitable, in method models they are used to express the project characteristics and what is required. Method patterns and method services are the actual building blocks to be used in method models and that are discoverable and composable via their interfaces.

MESP Method Model Regarding the second group, MESP Method models reference suitable method patterns and method services from the method repository. The referenced elements are composed using control and data flow in order to create consistent method models.

BPEL Process Model Regarding the third group, in order to execute method models, they are first transformed to equivalent BPEL/BPEL4People process models so that they can be executed with standard BPEL engines. Basically, each method service referenced in a method model is transformed to an equivalent HumanTask Invocation. The control and data flow information in the method model is used to create a BPEL Process model that properly connects the HumanTask invocations so that they are executed in the right order and with the right work products.

Method Enactment For the execution, additional work products are necessary: during the execution of the process model, whenever a HumanTask invocation is executed, a Workflow Task is created and assigned to the Person that is responsible according to her role. In addition, Runtime Information about the state of execution of the process model is maintained and logged.

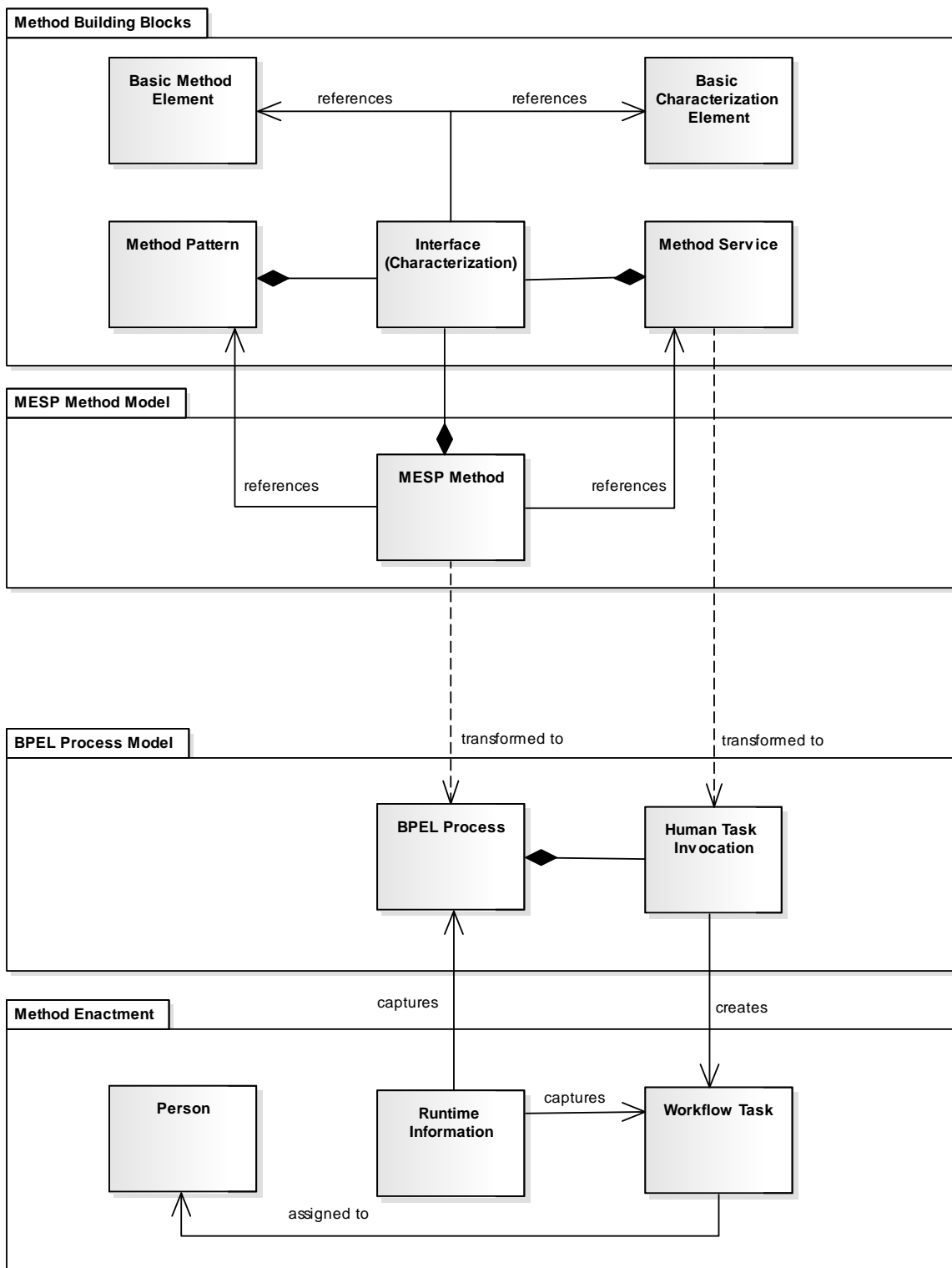


Fig. 3.3 Overview of central MESP Work Products

3.1.3 Overview of MESP Tools

Figure 3.4 illustrates the tools that are used as part of the MESP approach. The tooling consists of three groups. First, the MESP Tool that we created specifically for MESP. Second, an off-the-shelf Standard BPEL Engine with support for human workflow tasks. Third, an off-the-shelf Standard Project Repository.

MESP Tool Regarding the first group, the Method Building Block Editor is used by the senior method engineer to create method building blocks and to store them in the method repository. The project method engineer uses the Method Composer to compose methods out of building blocks. The created method models are also stored in the Method Repository. With the Consistency Checker, the project method engineer can resolve consistency issues of the method model. Consistent method models can be transformed with the MESP2BPEL Transformer into equivalent BPEL/BPEL4People process models to be executed with a BPEL engine. The Repository Browser can be used to browse through elements stored in the method repository.

Standard BPEL Engine Regarding the second group, the Standard BPEL Engine is used to execute process models and manage workflow tasks. Its Workflow Engine component executes the process model and request the creation of workflow tasks. These are managed within the Task Engine that interfaces with the project team members.

Standard Project Repository Regarding the third group, a Standard Project Repository is used to store the created work products and create URIs for them.

3.1.4 Integrated Overview of MESP Solution

Figure 3.5 illustrates the MESP solution with all the aspects discussed in this chapter: tasks, roles, tools, and work products. The top shows the method content definition layer, the middle the method tailoring layer, and the bottom the method enactment layer.

On the method content definition layer, reusable method content is extracted from documented methods, best practices, and also lessons learned from performed projects by the senior method engineer. This content is then formalized by basic elements on the one hand and method services and method patterns on the other. The elements are formalized using the building block editor and stored in the method repository.

On the method tailoring layer, each project is first characterized by the project method engineer using the method composer. Using the characterization, she com-

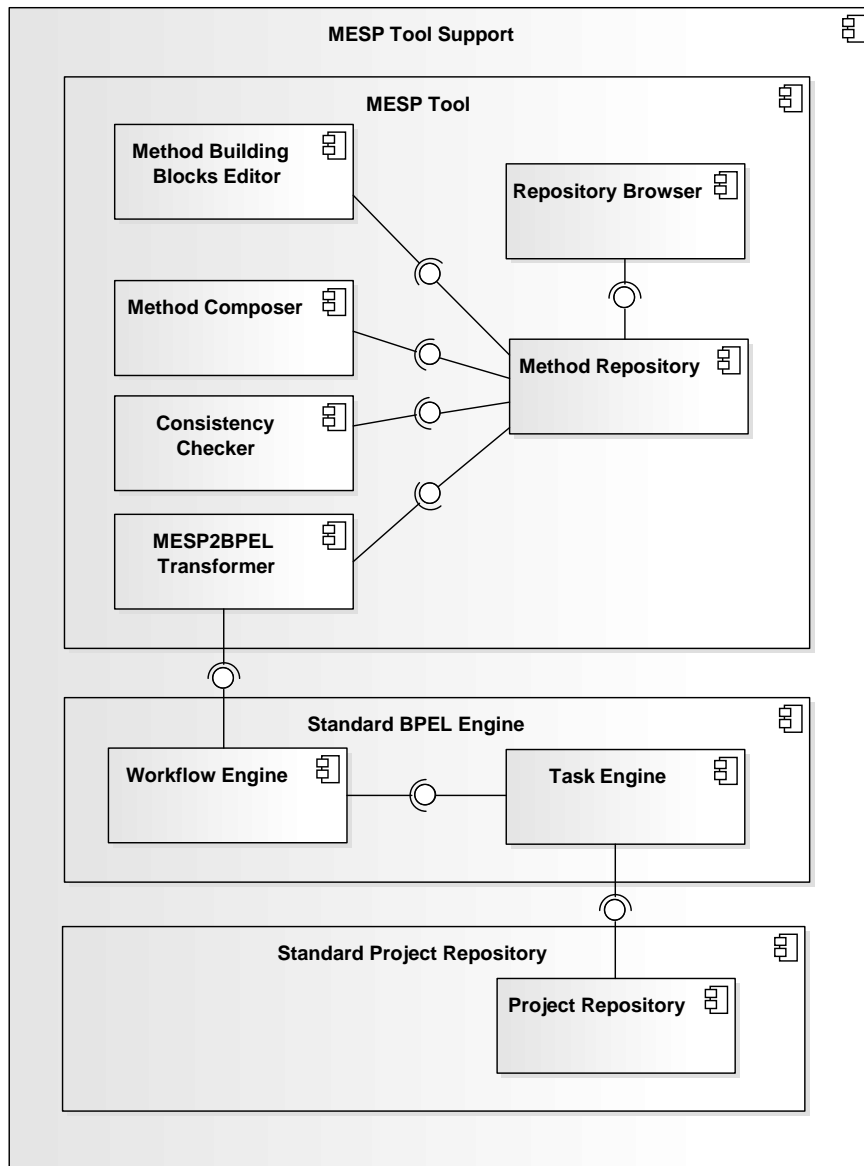


Fig. 3.4 Overview of MESP Tool Support

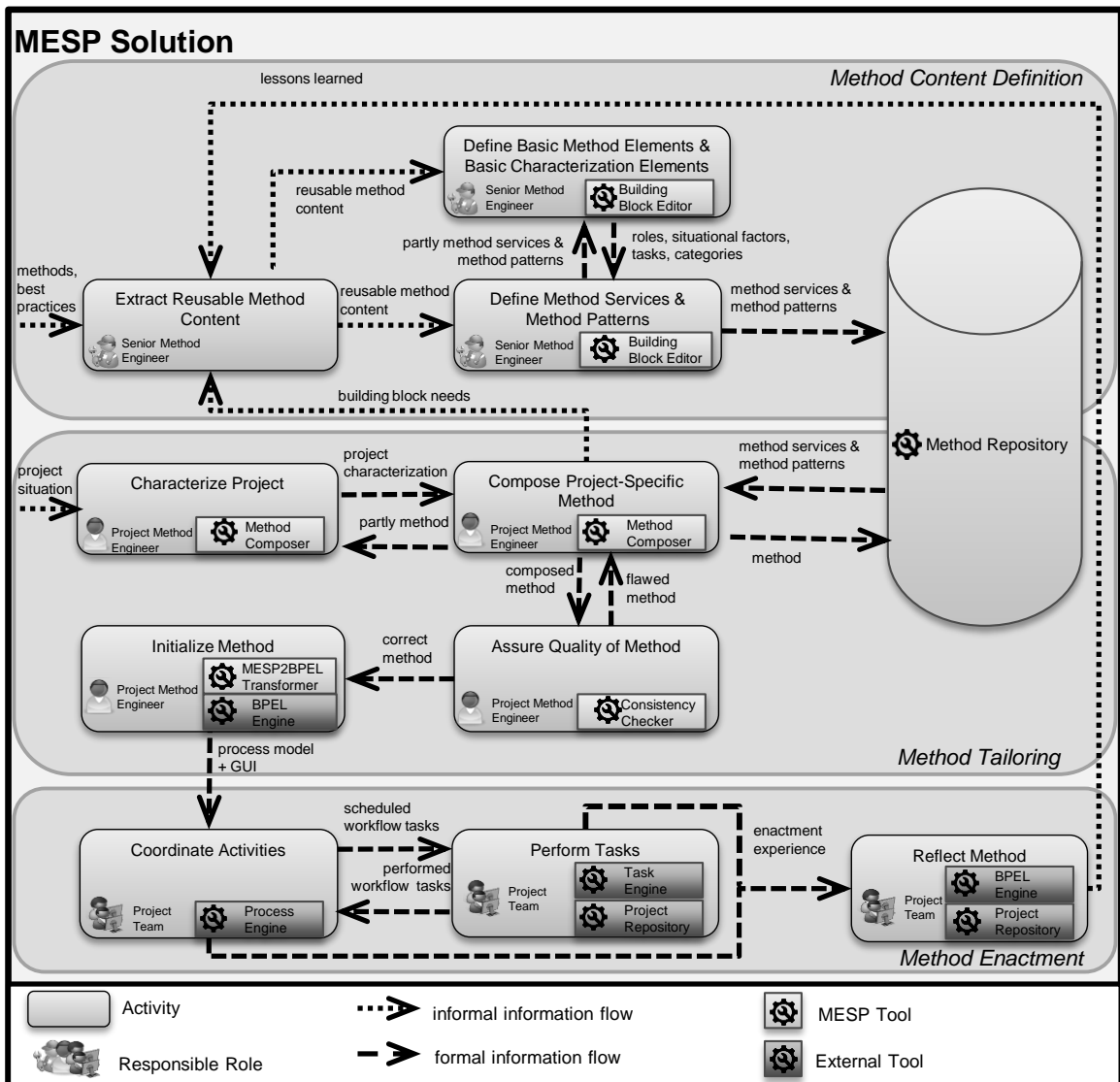


Fig. 3.5 An Overview of the MESP Solution Framework

poses a project-specific method and stores it into the method repository. Thereby, she can use the consistency checker to assure the quality of the method. Once the method model is consistent, she can initialize the method model for execution. She derives a process model with the MESP2BPEL Transformer, deploys it to the BPEL engine and assigns project members to roles.

On the method enactment layer, the process engine executes the process model and helps the project team to coordinate their activities. The task engine manages the performance of tasks and shows the responsible team members the task descriptions from the method model. After the project team reflects the method enactment and collects lessons learned, using the process execution information from the BPEL engine and the created work products from the project repository.

3.2 End-to-End Example

In the following, we provide an end-to-end example to illustrate our overall solution and in particular the tasks that we introduced in the previous section. Our example is a project from the eID domain as we described in [Faz+13] and deals with the introduction of a distributed ePassport system. Such a system is connected to several national (e.g. border control, civil register) and international (e.g. Interpol) databases and information systems.

Typically such a system is developed using a plan-driven approach, either a fixed off-the-shelf method or by creating a method using configuration-based SME, e.g., V-Modell XT [HHo8]. The project manager of such projects is typically a passport domain expert and not an experienced method engineer. Thus, when changes to a method are required, she is not able to tailor the method accordingly. One requirement might for example be to integrate agile aspects into the method to allow for stepwise refinement of the work products and to improve the communication among team members. The method has to be tailored such that it integrates meaningful agile aspects and still stays consistent. In addition, also its correct enactment has to be ensured.

Using this scenario, we provide an end-to-end example for the use of MESP in order to illustrate the benefits of our solution. We will briefly exemplify each MESP task in the context of the scenario. We also describe the relationship between work products and tools of each layer.

3.2.1 Method Content Definition

In the following, we illustrate the tasks of the senior method engineer to define reusable method content. As described, these tasks are carried out independently of individual projects as an ongoing, long-term effort to maintain a repository of up-to-date method services and method patterns. In our scenario, the senior

method engineer has to derive method content that is suitable to carry out eID projects with differing levels of customer involvement and system criticality.

Extract Reusable Method Content

With this task, the senior method engineer identifies and extracts reusable method content from literature and practice. This is a first step towards the formalization by creating new method services and method patterns. In the eID domain, projects are typically carried out with plan-driven methods following the v-model. Now agile aspects shall be incorporated to allow for stepwise refinement of work products. If possible, concrete guidance on how to refine these work products shall be given. In our example, the senior method engineer extracts content from the documented methods V-Modell XT, Scrum, and OpenUP to cover these requirements.

V-Modell XT The senior method engineer investigates the V-Modell XT as a software engineering method framework that creates plan-driven methods based on the v-model. Figure 3.6 illustrates the control flow of a V-Modell XT-based method that is denoted by its lifecycle. Each decision gate (left leaning parallelogram) marks the end of a lifecycle phase. At each decision gate, a set of documents has to be approved using a defined task "Project coming to a progress decision". For example, Figure 3.7 shows the documents that have to be approved for the decision gate *System Designed*. Consequently, these documents have to be produced with the designated V-Modell XT tasks before the decision gate can be passed. Thus, in methods based on V-Model XT, the sequence of decision gates indirectly specifies the order of activities that have to be performed.

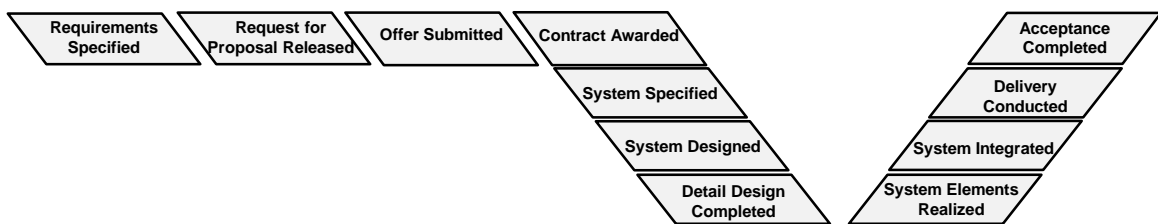


Fig. 3.6 The gateways of the V-Modell XT method

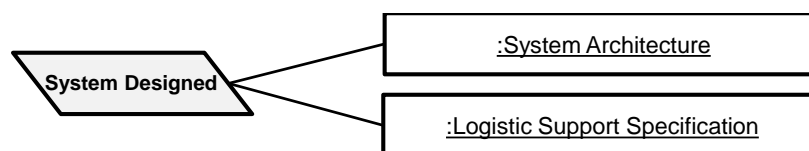


Fig. 3.7 The decision gate "System designed" of V-Modell XT

The first four decision gates include activities to determine the requirements of the system (decision gate: Requirements Specified), issue a tender (Request

for Proposal Released), receive offers (Offer Submitted), and select a supplier for the development of the system (Contract Awarded). The system is then specified according to the requirements specification (System Specified) and decomposed into components and subcomponents (System Designed, Detail Design Completed). Based on the design documents, the subcomponents are integrated to components (System Elements Realized) and components to the complete system (System Integrated). Finally, the system is integrated into the production environment (Delivery Conducted). Once the system is in place, it is tested and approved (Acceptance Completed).

Due to the popularity of the v-model in the eID domain, the coverage of the complete project lifecycle, and the systematic development and sign-off of formal documents, the senior method engineer decides to extract the V-Model XT lifecycle with its decision gates. In addition, she decides to extract the related tasks and work products.

Scrum In order to identify agile method content, the senior method engineer investigates the popular Scrum method [SS13]. Scrum is an iterative and incremental method, where the development is carried out in iterations of fixed length called *sprint*. Figure 3.8 illustrates the control flow of Scrum. A sprint basically prescribes a sequence of fixed length, where planning activities are followed by implementation activities, which in turn are followed by reviewing activities. The implementation activities are repeated daily until the end of the sprint and coordinated with informal, short meetings in the morning called *daily scrum*.

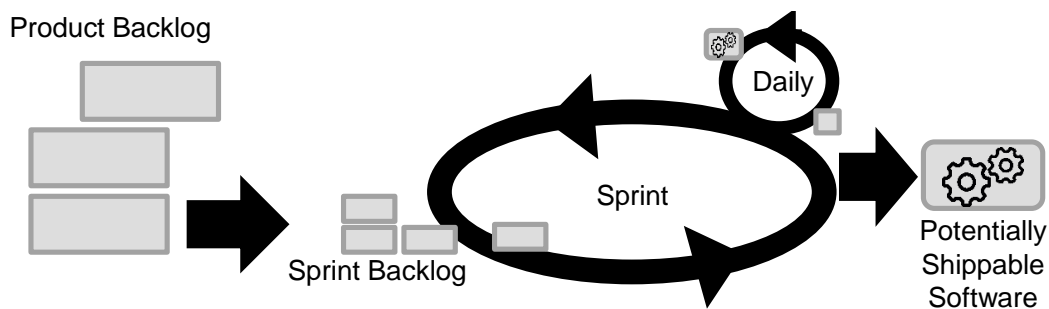


Fig. 3.8 Illustration of the process flow of Scrum

Due to the popularity of Scrum, its iterative and incremental nature, and its informal coordination meetings, the senior method engineer decides to extract the Scrum sprint loop. In addition, she decides to extract the related tasks and work products.

OpenUP In order to identify further helpful task descriptions that can be used in an iterative, incremental setting, the senior method engineer investigates the

OpenUP⁷ method. Among other tasks, she extracts the tasks to envision and refine a system architecture that she considers useful. Figure 3.9 shows an excerpt of the task *refine the architecture* from its specification website.

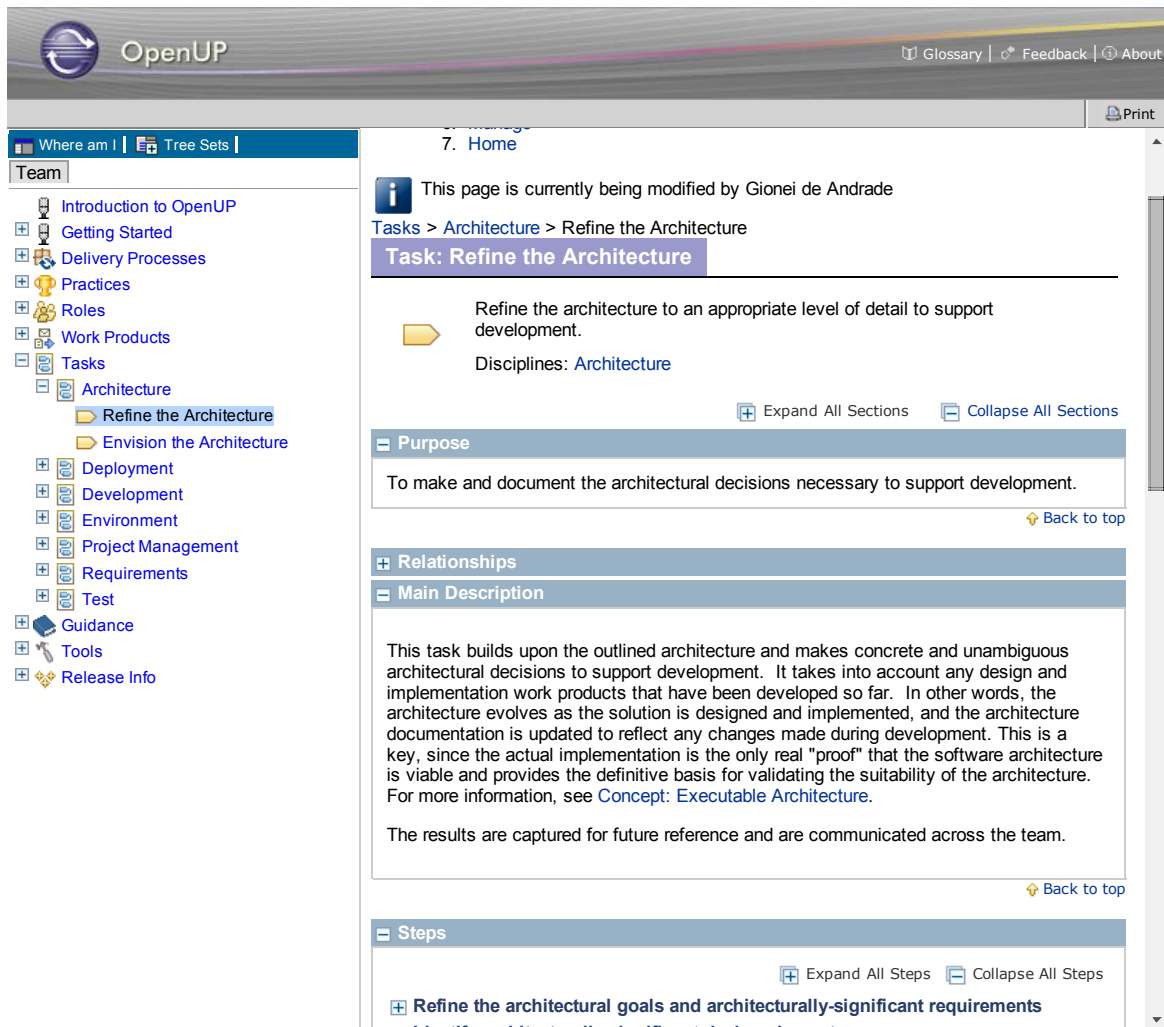


Fig. 3.9 Decription of the task "refine the architecture" of OpenUP

The MESP task *Extract Reusable Method Content* is carried out informally and without tool support as it is the preparation of formalizing method content.

Define Basic Elements

With this MESP task, the senior method engineer defines basic elements that are shared by *method services*, *method patterns*, and *method models*. There are two kinds of basic elements. *Basic method elements* are work products, roles and tasks. They are used to define the constituents of method services and they are also referenced

⁷<http://epf.eclipse.org/wikis/openup/>

in method patterns. *Basic characterization* elements are *situational factor values* and *categories*. They are used to characterize method services, method patterns and projects (and thereby the method to be composed). Figure 3.10 shows an UML object diagram that illustrates some basic elements and their relationships that were defined for the method content extracted for V-Modell XT, Scrum, and OpenUP in our scenario. We explain how they were derived in the following.

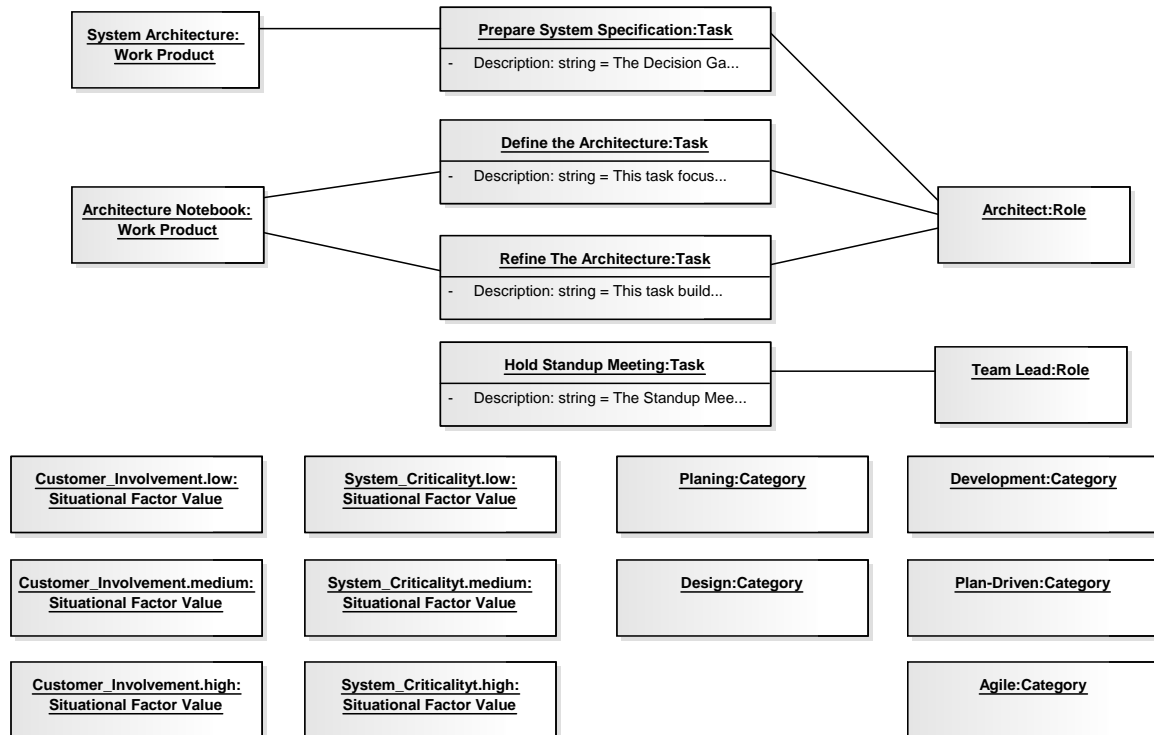


Fig. 3.10 Basic elements created by the senior method engineer

Basic Method Elements Regarding basic method elements, for example, the senior method engineer defines the task Prepare System Specification and the work product System Architecture for the extracted V-Modell XT decision gate shown in Figure 3.7. The required information is given explicitly in the V-Modell XT specification document. Instead of directly taking over the terminology used in V-Modell XT, she adopts it to fit to the existing basic elements in the method repository. Thus, instead of defining a role System Architect, she defines the role Architect that is also responsible for tasks derived from OpenUP. The V-Modell XT decision gate also defines tasks, roles, and work products for the logistic support. However, the senior method engineer sees no use for them right now and does not add them.

For the content from Scrum, she creates the task Hold Standup Meeting that reflects the daily scrum meeting of Scrum. Again, she rephrases the terminology.

As responsible role for the task, she defines a Team Lead role that did not exist yet. It is an adoption of the role Team that is described in the Scrum specification document.

For the content from OpenUP, she creates the tasks Define the Architecture and Refine the Architecture, as well as the work product Architecture Notebook. Here, she can reuse the already defined role Architect.

Basic Characterization Elements Regarding basic characterization elements, *situational factor values* determine, in which context a method building block should be included in the method and in which context not. These are usually not stated explicitly in the extracted method content and need to be derived based on secondary literature (e.g. [Bek+08] or [CO12]) and project experience. For example, the senior method engineer in our scenario defines the situational factor Customer_Involvement as she learns that in Scrum a lot of customer involvement is required, while in V-Modell XT it is required only selectively at some of the decision gates. She defines the three situational factor values Customer_Involvement.low, Customer_Involvement.medium, and Customer_Involvement.high. Situational factor values will be used later to characterize whether customer feedback can be given frequently in a project and whether it is required by a method building block.

Another situational factor the senior method engineer defines is the System_Criticality as she knows that for critical systems certain plan-driven tasks are required, while they might be unsuitable for uncritical systems. Here, she defines the situational factor values System_Criticality.low, System_Criticality.medium, and System_Criticality.high.

Categories are an additional means to characterize method building blocks independent of a project situation, for example by origin, discipline, or typical phase. This is used to describe useful method services for a method pattern. In our scenario, the senior requirements engineer for example creates the categories Agile, Plan-Driven, Development, Design, and Planning based on the method content she extracted.

The basic elements define the formal terminology that is used to describe method services, method patterns, and methods. They therefore influence which method services and method patterns are interoperable.

The MESP task *Define Basic Elements* is supported with tooling. Basic elements are created with the *Method Building Block Editor* and stored in the *Method Repository*. The Method Building Block Editor offers checks to ensure that the modeled elements are specified correctly and adhere to the MESP meta-model.

Define Method Services & Method Patterns

With this MESP task, the senior method engineer defines the reusable method building blocks that can be used to tailor project-specific method models. In our

scenario, based on the extracted method content and the basic elements, she creates method services and method patterns for V-Modell XT, Scrum, and OpenUP.

Method Services In our scenario, the senior method engineer first creates a method service for the task *refine the architecture* from OpenUP. For this, she reuses the respective task element that she created, as described in the previous section. The method service wraps the task and associates it with suitable basic characterization elements that help to discover and compose it. For example, this method service shall be used in methods, where the system to be developed is somewhat critical (*medium – high*), otherwise it bears unnecessary overhead. In addition, it is suitable, if there is a certain level of customer involvement planned in the project (*medium*), as decisions need to be supported by the customer. If there a lot of customer involvement is available, however, the method service is less suitable than other (agile) alternatives. Figure 3.11 shows an UML object diagram of the created method service. It references several basic elements from the method repository that were defined before. As basic method elements, it references the task, with its role and its work products. As basic characterization elements, it references the situational factor values *customer_involvement.medium*, *system_criticalityt.medium*, and *system_criticalityt.high*.

In a similar manner, the senior method engineer defines method services for the other tasks elements she created. Once she is done, she has created a set of method services to describe the necessary tasks for eID projects with different characteristics.

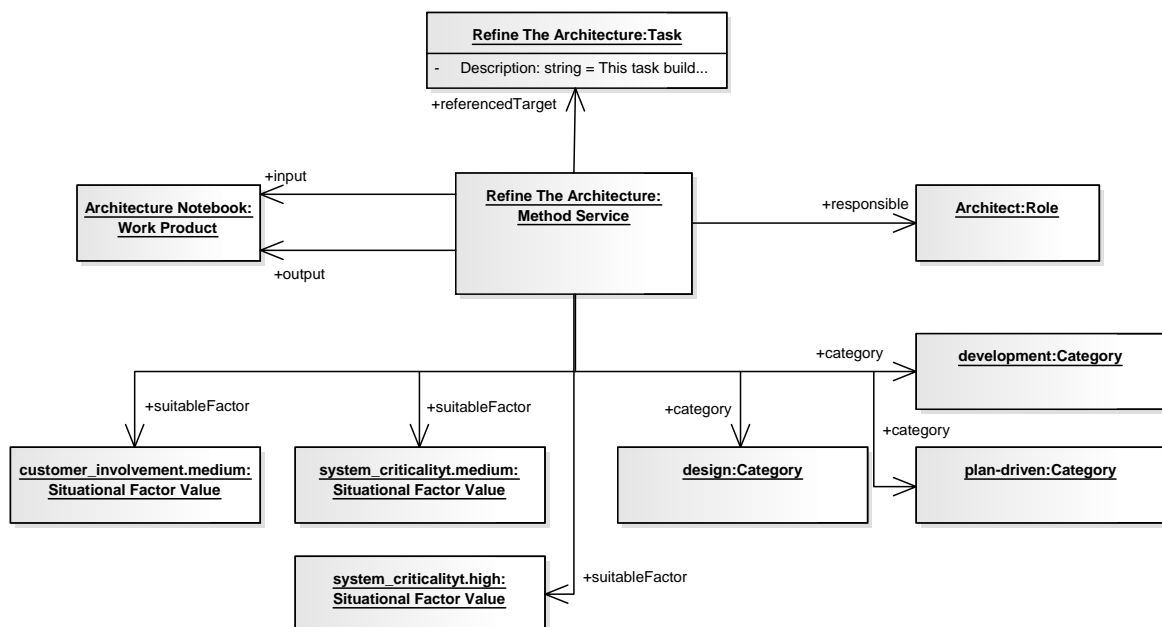


Fig. 3.11 A method service derived from OpenUP referencing basic elements

Method Patterns In addition to method services, the senior method engineer creates method patterns. For example, the senior method engineer wants to formalize the sprint from Scrum so that it can be used by project method engineers in their method models. As described previously, a sprint is mainly a time-boxed sequence where planning activities are followed by implementation activities, which in turn are followed by reviewing activities. If this was modeled with a method service, other method services could not be added into the Sprint during method composition. Thus, this would be very inflexible. So instead, the senior method engineer models a method pattern as described in the following.

She wants to model three groups of planning, development, and reviewing activities that are executed repeatedly. Figure 3.12 illustrates the method pattern that she creates. It consists of three sequential constrained scopes, one for each group of activities, which are executed inside a loop. Each constrained scope contains a constraint and space where method services can be put in later when the pattern is used. The space of a constrained scope has to fulfill the constraint. For example, the middle constrained scope requires that all method services placed inside are of category development and that there is one method service that is the method service *Hold Standup Meeting*. Constraints are expressed with a specialized domain specific language created for MESP that we describe in more detail in Chapter 4.

Like method services, method patterns are characterized with basic elements. This method pattern is associated with the situational factor values `Customer_Involvement.medium` and `Customer_Involvement.high` as the customer is involved in the planning that takes place fairly often. In addition, it is associated with `System_Criticality.low` and `System_Criticality.medium`, because it is very challenging to apply the pattern to create correct critical systems [SA07]. The pattern is furthermore associated with the category *Agile*. Figure 3.13 shows an UML object diagram that illustrates the relationship between the method pattern and the basic elements in the method repository. As shown, the constraints are not expressed in plain natural language, but reference the according elements in the method repository. For example, the constraint in the middle constrained scope references the category `Development` and the method service `Hold Standup Meeting`.

The senior method engineer also creates a pattern that represents the decision gates defined by V-Modell XT. It can be used by project method engineers to ensure that the method model they compose includes all tasks in order prescribed by the decision gates. Similar to the sprint loop method pattern, it consists of a sequence of constrained scopes that contain constraints. The resulting method pattern contains a constrained scope for each decision gate as illustrated in Figure 3.14.

We use the decision gate *System Designed* of Figure 3.6 as an example to show, how a constrained scope is derived. The resulting constrained scopes are illustrated

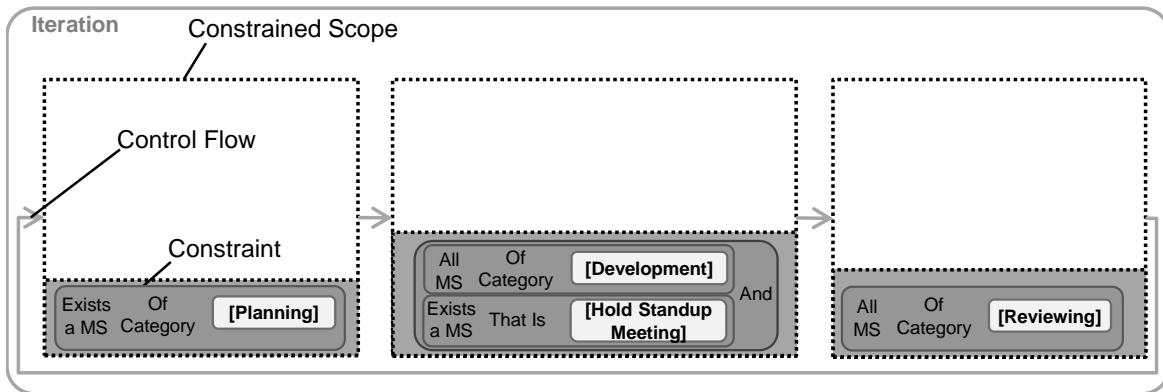


Fig. 3.12 A method pattern based on the Sprint from the Scrum method

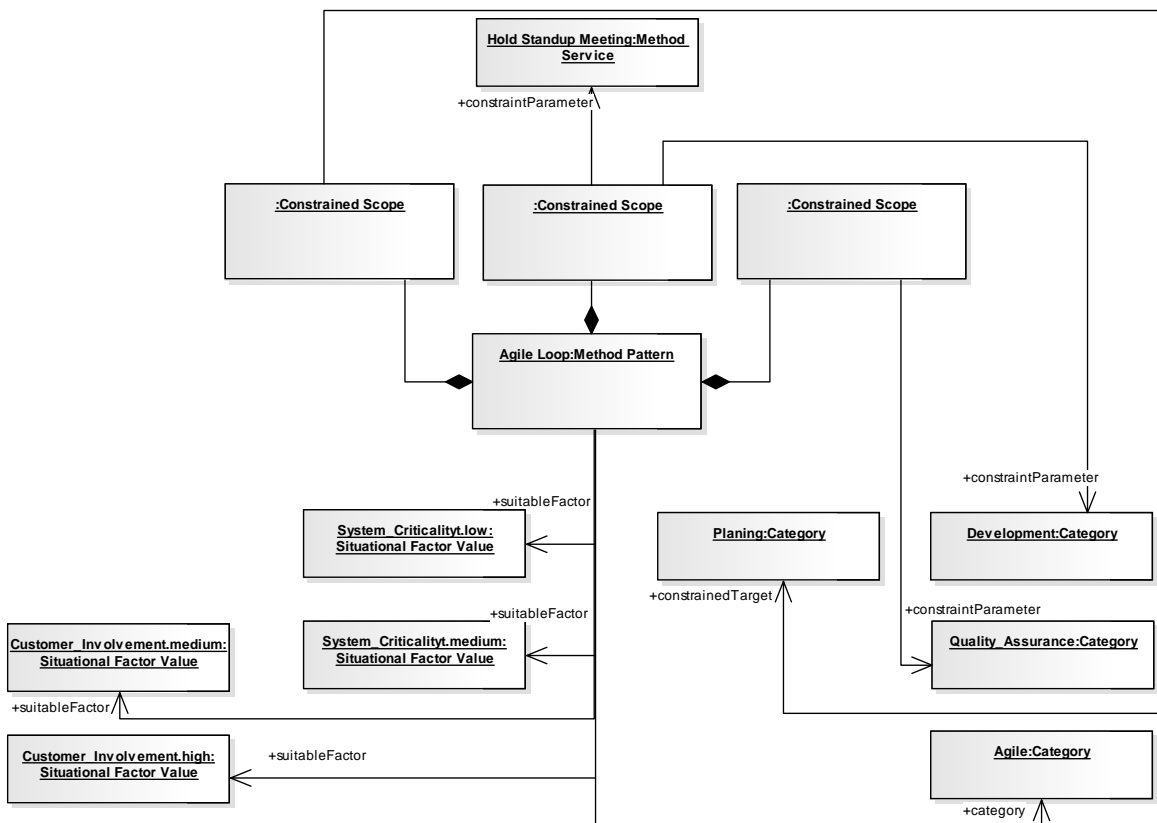


Fig. 3.13 The object model of the method pattern based on the Sprint from the Scrum method

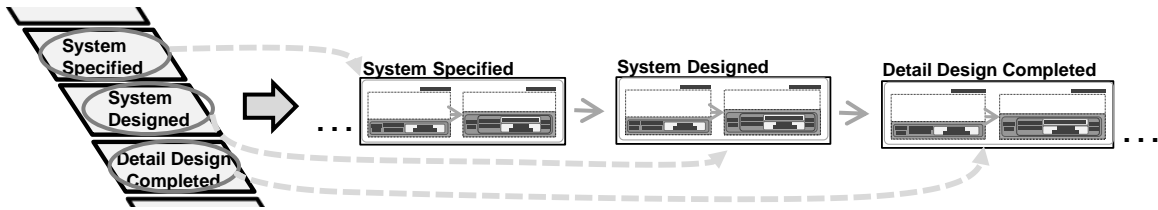


Fig. 3.14 The method pattern for the V-Modell XT is derived by creating constrained scopes for each decision gate

in Figure 3.15. The constrained scopes ensure that the relevant work product *system architecture* is created and assessed afterward. The senior method engineer left out the other work products of the decision gate, which are irrelevant for the eID projects of her organization.

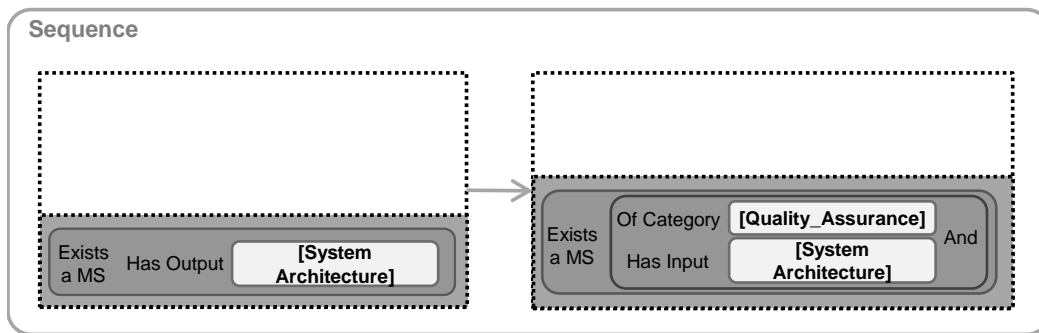


Fig. 3.15 Constraint scopes that reflect the decision gate “System Designed” of V-Modell XT

The senior method engineer associated the resulting method pattern with the situational factor values *Customer_Involvement.low* and *Customer_Involvement.medium* as the customer is involved only in specific points in time and fairly limited. In addition, she decides to associate the situational factor values *System_Criticality.medium* and *System_Criticality.high*, because the pattern is suitable to build critical systems, but results in a process and documentation overhead unnecessary for uncritical systems.

The two method patterns allow combining plan-driven and agile aspects later during method composition as required by some eID projects. By creating the V-Modell XT lifecycle pattern, the senior method engineer helps to ensure the plan-driven nature of method models as required for eID projects. The sprint pattern expresses the necessary aspects of an agile development cycle.

The MESP task *Define Method Services & Method Patterns* is supported with tooling. Method patterns and method services are created with the *Method Building Block Editor* and stored in the *Method Repository*. Like for basic elements, the Method Building Block Editor offers checks to ensure that the modeled method services and method patterns adhere to the MESP meta-model.

Work Products and Tools

Figure 3.16 shows an UML class diagram that gives an overview of the relationship of the MESP work products for method content definition. We can group basic elements into Basic Method Elements and Basic Characterization Elements. As illustrated, Method Patterns and Method Services reference basic method elements via an explicit Interface. The interface helps to bundle those references. The Constrained Scopes of method patterns reference basic elements as part of their constraints.

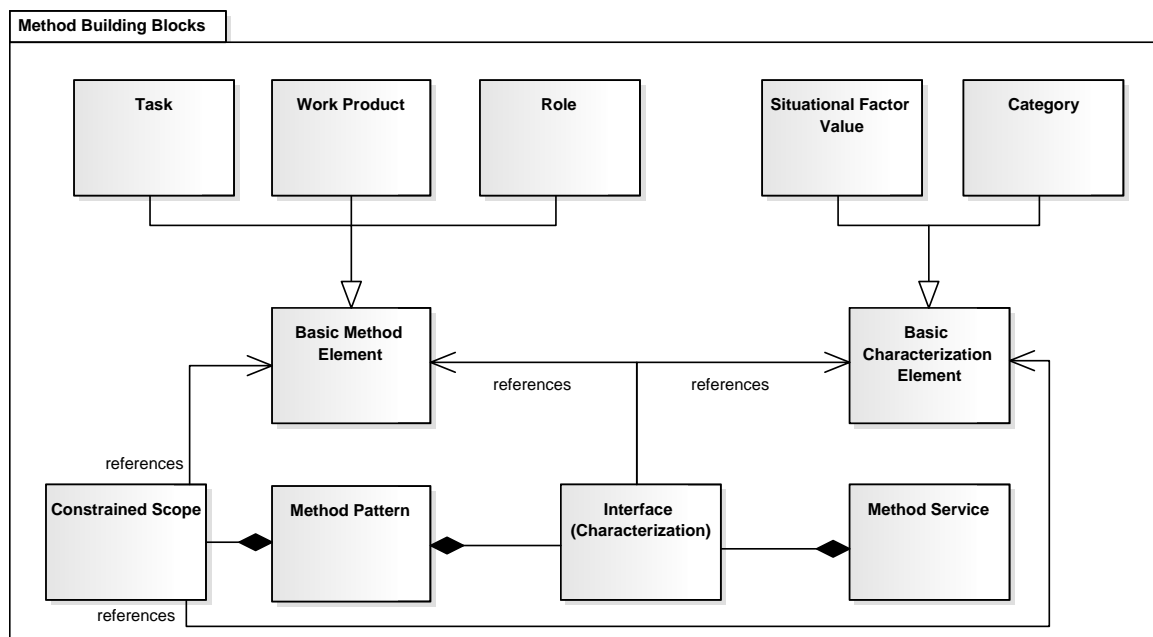


Fig. 3.16 Relationship of the Work Products for Method Content Definition

Figure 3.17 shows an UML component diagram that gives an overview of the relationship of the MESP tools used to create the MESP work products for method content definition. As explained, the Method Building Block Composer is used to create basic elements, method services, and method patterns and to store them into the Method Repository. The Repository Browser can be used to browse through the created elements and to invoke their editing with the Method Building Block Composer.

3.2.2 Method Tailoring

In the following, we illustrate the tasks of the project method engineer to compose project-specific, situational method models using existing method services and method patterns from the method repository. In our scenario, a project method engineer needs a suitable method for an eID project with a new client that wants to

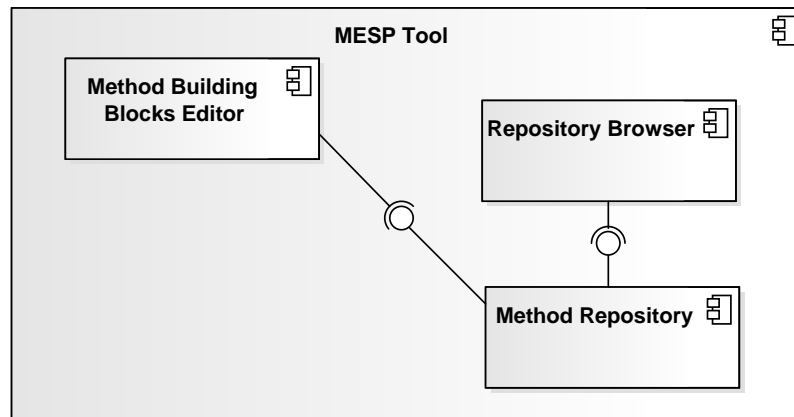


Fig. 3.17 The MESP tools to define method content

be involved throughout the development. An ePassport system shall be developed that is important to the client, but does not pose a risk to human lives. The project manager Alice has to propose the method that should be followed. She is familiar with the basic principles of plan-driven and agile methods, but has no in-depth knowledge of method engineering.

Characterize Project

With this MESP task, the project method engineer formalizes the project characterization using the available basic elements in the method repository. She characterizes the project in terms of the *project goal* and the *project situation*. In our scenario, Alice has to characterize the ePassport project with the new client.

Project Goal The project method engineer determines the project goal by defining which work products defined in the method repository are available at project start and which work products are to be delivered at the end of the project. In our example, a *requirements specification* has already been created by the client and is available at the start of the project. Required outputs are an *implementation* and an *integration test result* to demonstrate that the implemented system works as intended.

Project Situation The project method engineer determines the project situation by defining what situational factor values are suitable for the project. In our ePassport project, the system is somewhat critical and the customer is available on a frequent basis, but not continuously. Thus, Alice selects the situational factor values *system_criticality.medium* and *customer_involvement.medium*.

Figure 3.18 shows an UML object diagram of the project characterization that Alice created for the project. She associated the Project Goal and Project Sit-

uation of the MESP Method with the respective basic elements from the method repository.

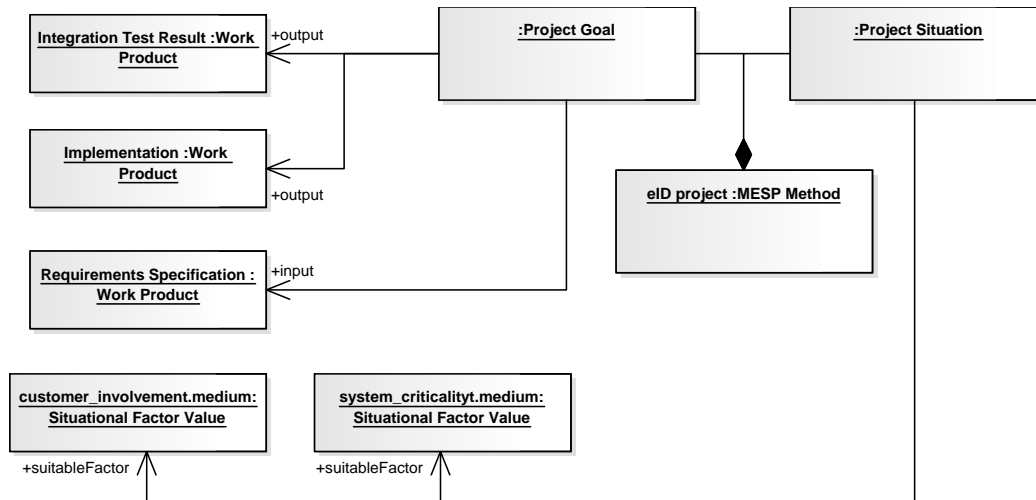


Fig. 3.18 The characterization of a project with basic elements from the method repository

Characterizing the project with project goal and project situation supports the following MESP task of discovering suitable method services and method patterns. In addition, it documents the project context for future reference, e.g., when reflecting the method enactment.

The MESP task *Characterize Project* is supported with tooling. The project characterization is created with the *Method Composer* that allows accessing and referencing available basic elements from the *Method Repository*.

Compose Project-Specific Method

With this task, the project method engineer identifies suitable method services and method patterns and composes them to a method model. Composing the method solely using method services would not offer sufficient guidance to allow less experienced project method engineers like Alice. Therefore, method patterns are used to restrict the potentially composable methods to meaningful ones. Afterward, method services are added to the method.

Method Patterns Method patterns are selected and combined to a frame that hosts potentially suitable method services. In our scenario, based on the situational factor values of the project situation, *Customer_Involvement.medium* and *System_Criticality.medium*, Alice identifies the method patterns derived from Scrum and V-Modell XT. As presented in the previous section, the method patterns are associated with the same situational factor values. For each pair of constraint scopes that reflect a decision gate of V-Modell XT, Alice adds a sprint loop method

pattern derived from Scrum, so that the work will be executed in an agile manner. Figure 3.19 exemplifies the combination using the constrained scopes for the system designed decision gate. As shown, the sprint pattern was added into first *Specification* constrained scope, where the *system architecture* work product has to be created. As a result, the method services to be added have to fulfill both, create a system architecture and development in an agile manner.

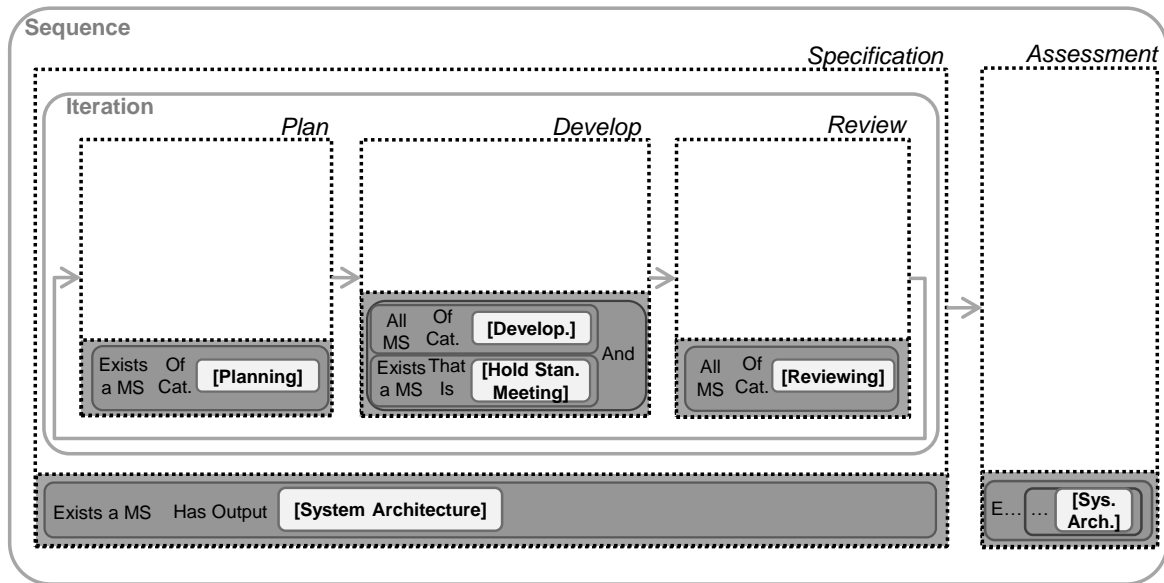


Fig. 3.19 Two method patterns combined to one pattern

Method Services Suitable method services are identified based on the project characterization, the method pattern constraints, and already chosen method services. Regarding the project characterization, for example, among the required outputs of the project is the *integration test result*. Thus, Alice has to add at least one method service to the method model that provides this output. In addition, searching for suitable method services with the situational factor values of the project brings up the earlier described method service *Refine The Architecture*.

Regarding the method pattern constraints, for example, the *Specification* constrained scope in Figure 3.19 requires a method service that creates a system architecture work product. In addition, the contained *Develop* constrained scope requires the use of the method service *Hold Standup Meeting*.

Regarding already chosen method services, for example, the method service *Refine the Architecture* requires an architecture notebook artifact as input. Looking for method services that produce an architecture notebook brings up the method services *Envision the Architecture*.

Alice adds the suitable method services into the method model. Thereby, she pays attention to the constraints of the method patterns. Figure 3.20 shows the

partially composed method model for the ePassport project. As the method services *Hold Standup Meeting*, *Refine the Architecture*, and *Prepare System Specification* are associated with the category *Development* and the method service *Hold Standup Meeting* was added, the constraint of the *Develop* constrained scope is fulfilled. By adding further method patterns and method services, Alice composes the method step by step until all output work products required by the project goal are created by methods services and all constraints of the used method patterns are fulfilled. Thereby, she composes a method model that is specific to the context of her ePassport project.

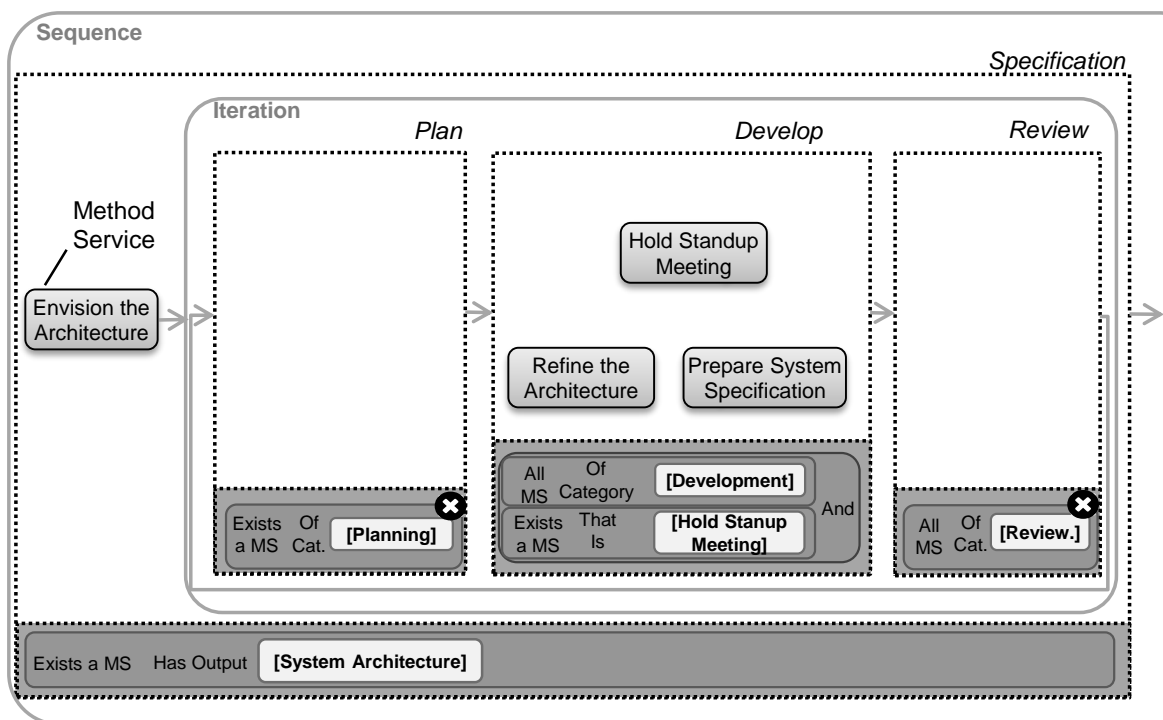


Fig. 3.20 A partial method model with method services

Control Flow & Data Flow In order to compose a unambiguous method model, control and data flow information is added to the method model. For example, in Figure 3.20 is not clear, whether the three method services in the *Develop* constrained scope shall be enacted sequentially or in parallel. In addition, the data flow of work products is not specified, yet. If there were multiple method services with the same output, it would not be clear which input to take. Figure 3.21 shows a more advanced state of the method model composition. Here, Alice added additional control flow for the *Develop* constrained scope. The method service *Hold Standup Meeting* takes place repeatedly in parallel to the sequentially enacted method services. In addition, Alice specified the data flow between *Envision the*

Architecture and *Refine the Architecture*. However, the method model is still not finished, as denoted by the ☒.

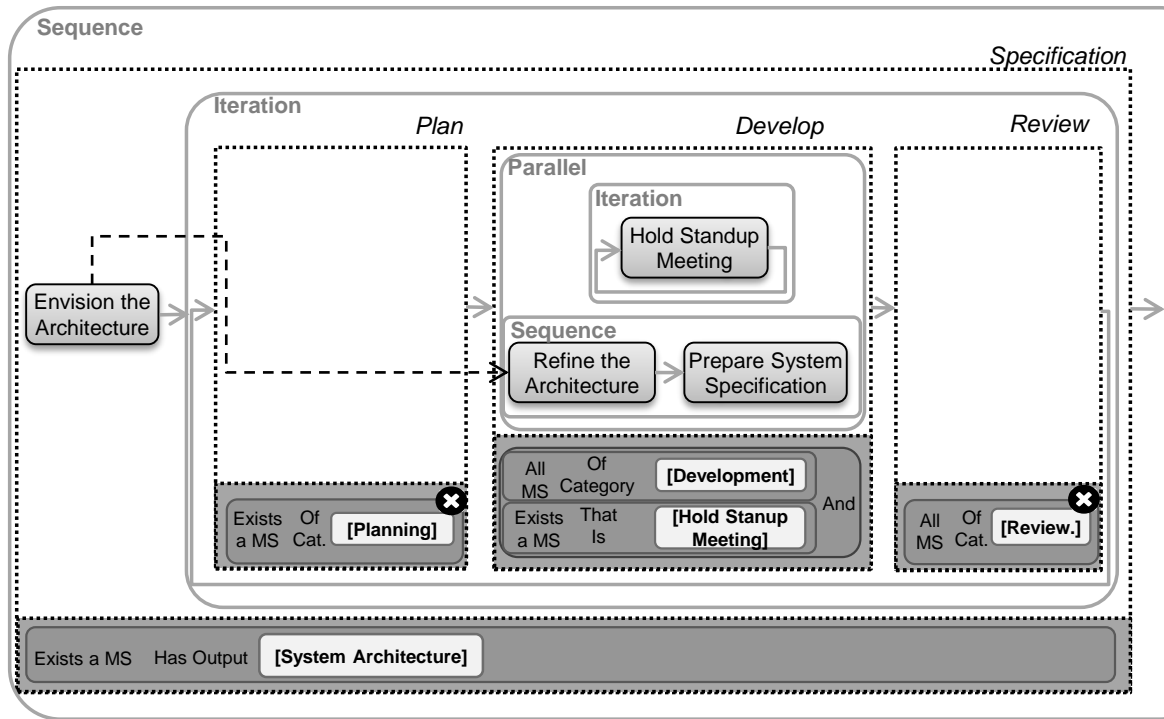


Fig. 3.21 A partial method model with additional control and data flow

The MESP task *Compose Project-Specific Method* is supported with tooling. The method model is composed with the *Method Composer* that allows adding method patterns and method services from the *Method Repository* and to specify additional control and data flow.

Assure Quality of Method

With this task, the project method engineer checks the composed method model for completeness and other quality characteristics. For example, she checks, whether all required input work products are provided for all method services and whether all method pattern constraints are fulfilled. The method model has to be free of certain quality issues in order to be executable with a process engine later during method enactment.

For example, when Alice checks the partial method model in Figure 3.21 she finds out that the constraints for the constrained scopes *Plan* and *Review* are not fulfilled yet as indicated by the ☒. To detect these and similar issues, Alice can use the quality analysis that we implemented as part of our tooling. In order to remove the issues, Alice searches for suitable method services and adds them to the method model. In some cases, this might require the refinement of the

project characterization. Thus, the tasks to characterize the project, compose the project-specific method, and to assure the quality of the method are carried out incrementally. Figure 3.22 shows the corrected partial method model. By adding the two method services *Iteration Planning Meeting* and *Iteration Review Meeting* and the data flow between them, all method patterns are fulfilled, and the data and control flow is specified.

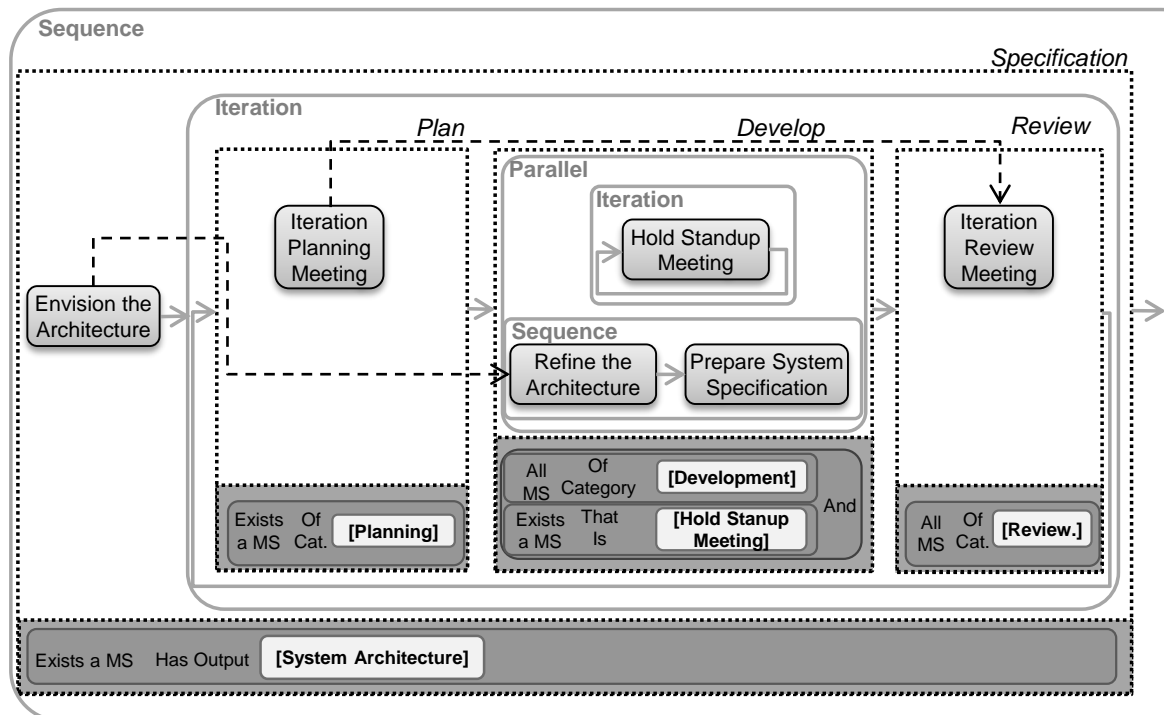


Fig. 3.22 A correctly specified partial method model

After removing all issues from the method model, Alice has created a consistent method model for her ePassport project, where the method model combines method patterns and method services from V-Modell XT, Scrum, and OpenUP.

The MESP task *Assure Quality of Method* is supported by an analysis component, as described. The *Consistency Checker* helps the project method engineer to detect and resolve quality issues in her method model.

Initialize Method

With this task, the project method engineer prepares the method for enactment by the project team. She derives a process model that can be executed with a standard BPEL/BPEL4People Engine and assigns project team members to roles used in the method. The process model contains only concepts of BPEL/BPEL4People, thus, each concept of the MESP method model has to be transformed appropriately. For example, each method service has to be transformed into an appropriate

BPEL4People task for the team member, a *HumanTask Invocation*. We implemented this automated transformation as part of our tooling.

In our scenario, after Alice finished the method model she invokes the automated transformation. The transformation creates the deployment file with the BPEL/BPEL4People process model and deploys it into the BPEL engine. Alice then uses the configuration interface of the BPEL engine to assign her team members to the roles used in the ePassport method. For example, she assigns the architect role to Bob and the team lead role to herself. Figure 3.23 shows a screenshot of the configuration interface of the BPEL engine. After the process model is deployed and the roles are assigned, Alice starts the process and enters the location URI of the requirements specification that is available at project start. The process model is then ready for execution.

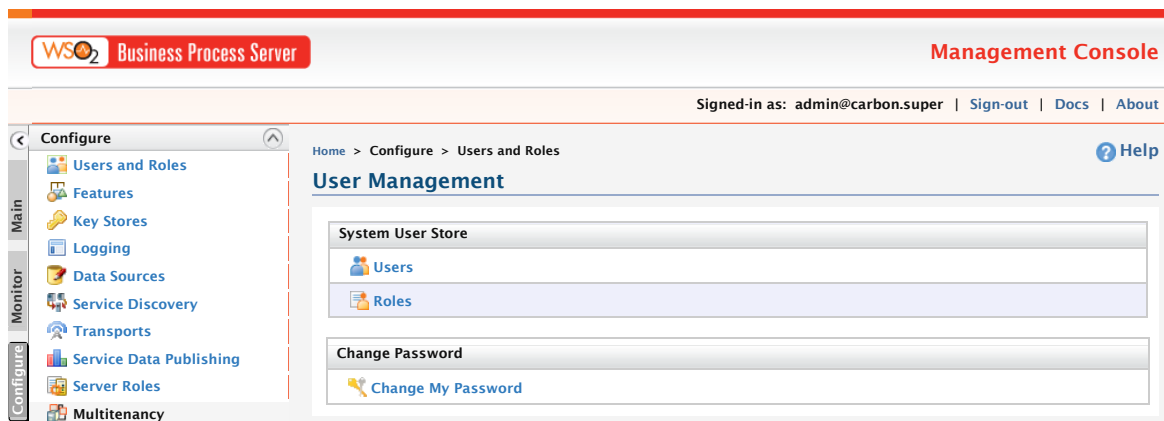


Fig. 3.23 Configuration Interface of the BPEL engine to set up users and roles

The MESP task *Initialize Method* is supported by a transformation component, as described. The *MESP2BPEL Transformer* transforms the MESP method model to a BPEL/BPEL4People process model and deploys it to the *Standard BPEL Engine*, the execution environment of the process model.

Work Products and Tools

Figure 3.24 shows an UML class diagram that gives an overview of the relationship of the MESP work products for method tailoring. As explained, the project Characterization consists of the Project Goal that references required and provided work products and the Project Situation that references situational factor values. The project characterization is part of the MESP method model and can be seen as its interface as it describes inputs and outputs of the method and the nature of the project. The MESP Method references method patterns and method services from the project repository that are used in the method. The MESP method is transformed to a BPEL Process and especially BPEL4People HumanTask Invocations that are derived from the Method Services.

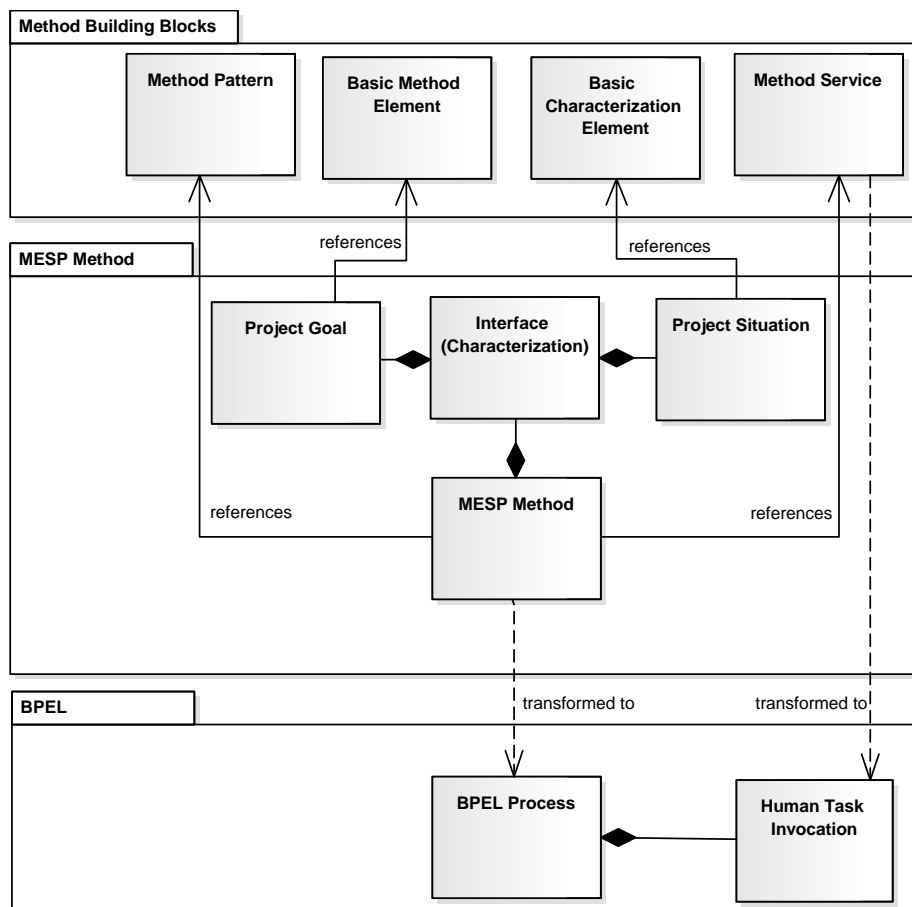


Fig. 3.24 Relationship of the MESP Work Products for Method Tailoring

Figure 3.25 shows an UML component diagram that gives an overview of the relationship of the MESP tools used to create the MESP work products for method tailoring. As explained, methods are composed with the *Method Composer*. Created methods are stored in the *Method Repository*. With the *Consistency Checker*, the project method engineer can analyze the quality of the composed methods and with the *MESP2BPEL Transformer*, she can transform MESP method models to BPEL/BPEL4People process models. These models can be deployed into a *Standard BPEL Engine* for their execution that supports the project team in enacting the method.

3.2.3 Method Enactment

In the following, we illustrate the tasks of the project team to enact the method as prescribed and to provide feedback for method content improvement. In our scenario, the project team, Alice, Bob and their colleagues have to enact the method defined specifically for their ePassport project. In addition to the tasks from the

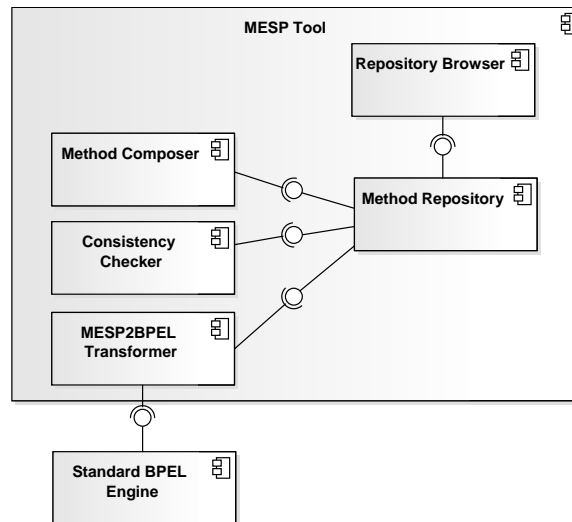


Fig. 3.25 The MESP tools for Method Tailoring

V-Modell XT that they are familiar with, the team has to carry out the novel tasks derived from Scrum and OpenUP.

Coordinate Activities

With this task, the project team coordinates what activities need to be carried out and what input work products to use. In particular, they have to ensure that they carry out the activities in the right order, that they notify each other when an activity is finished, and that they inform each other, where to find the output work products. The BPEL engine takes care of this task completely and coordinates the activities for the project team as illustrated in Figure 3.26. The BPEL engine executes the process model and whenever an activity is executed, it creates an appropriate workflow task (1). The assigned team member can access the workflow task via the BPEL engine (2) and mark it as finished, when it is done (3,4). Then, according to the control flow in the process model (5), the BPEL engine executes the following activity and creates a workflow task for it (6) and so on (7-10).

In our scenario, Alice just finished her workflow task to host the *Iteration Planning Meeting*. According to the control flow of the method model (see Figure 3.22), the BPEL engine creates two workflow tasks, one for the method service *Refine the Architecture* and one for the method service *Hold Standup Meeting*. It assigns the first workflow task to Bob, as he is assigned the architect role. Once Bob performed his workflow task, the BPEL engine invokes the following method service *Prepare System Specification* and creates another workflow task for him. Once he finishes his workflow task and the iteration with *Hold Standup Meeting* is finished, the BPEL engine invokes the next activity *Iteration Review Meeting*. It creates the according workflow task and assigns it to Alice, who is assigned the team lead role.

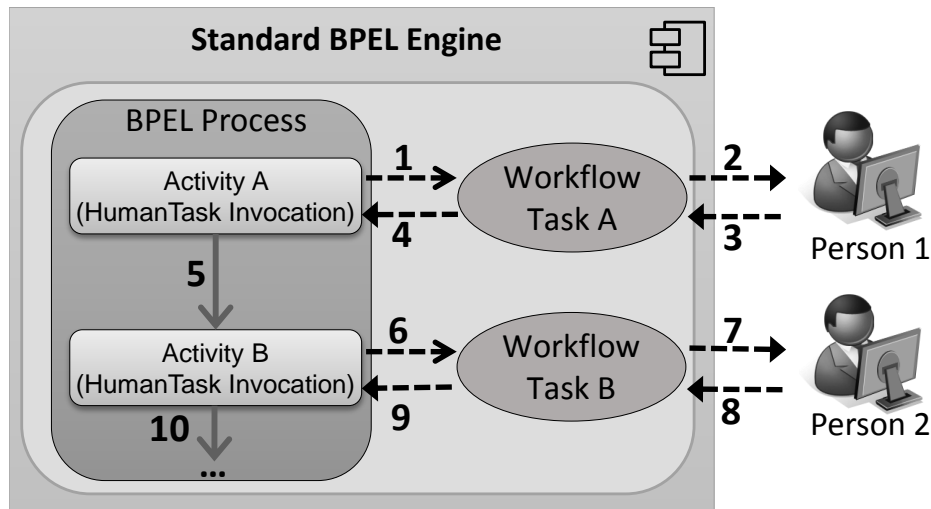


Fig. 3.26 The coordination of activities via workflow tasks

The MESP task *Coordinate Activities* is supported by the *Standard BPEL Engine*. As described, it executes the BPEL/BPEL4People process model and creates appropriate workflow tasks.

Perform Tasks

With this MESP task, members of the project team carry out the actual tasks defined within the method services of the method model. The responsible team member has to perform the task according to its description and by using the proper input work products. After finishing the task, the team member has to notify the teammates and inform them about the created output work products.

Like the previous MESP task, this MESP task is also supported by the BPEL engine. As explained in the previous section, the BPEL engine creates appropriate workflow tasks for invoked activities. A project team member can access her due workflow tasks via the task engine that is part of the BPEL engine. As part of our automated transformation from MESP method models to BPEL/BPEL4People, each workflow task contains a description taken from the according task in the method repository. In addition, each workflow task shows the location URIs of the input work products and offers fields to enter the output work product URIs. It also shows general information about the state of the executed process, e.g., the number of the current iteration. As part of the standard task engine interface, the team member can mark a task as completed. The BPEL engine then invokes the following activity according to the control flow.

In our scenario, Bob has to perform a workflow task for the method service *Refine the Architecture*. As shown in Figure 3.27, he is presented the description that corresponds to the original task description in the method repository (cf. Figure 3.9).

He also sees the location of the input Architecture Notebook in the Redmine⁸ ticket system that they the team uses as a project repository. After he uploaded his created output, he adds its URI to the workflow task. Once he marks his workflow task as finished, workflow tasks for the following activities will be created. So naturally, the MESP tasks to coordinate activities and to perform tasks alternate during method enactment.

[<< Back to Task List](#)

Details:	
Created On:	Mon Mar 26 14:04:03 CET 2016
Updated On:	Mon Mar 26 15:27:01 CET 2016
Status:	IN_PROGRESS
Description:	
To make and document the architectural decisions necessary to support development.	
People:	
Owner:	Bob
Request:	
Name	Refine the Architecture
Description	This task builds upon the outlined architecture and makes concrete and unambiguous architectural decisions to support development. It takes into account any design and implementation work products that have been developed so far. In other words, the architecture evolves as the solution is designed and implemented, and the architecture documentation is updated to reflect any changes made during development.
Role	Architect
Steps	Refine the architectural goals and significant requirements Identify architecturally significant design elements Refine architectural mechanisms Define development architecture and test architecture Validate the architecture Communicate decisions
Add. Roles	None
Inputs	Architecture Notebook : http://redmine.s-lab.de/issues/15
Outputs	Architecture Notebook
Phase	Development Phase
Iteration	Development Iteration: 1
Response:	
Architecture Notebook:	<input type="text"/>
<input type="button" value="Complete"/>	

Fig. 3.27 A workflow task for the method service Refine the Architecture

The MESP task *Perform Tasks* is supported by the *Standard BPEL Engine* and a *Standard Project Repository*. The BPEL engine contains two components: the *Task*

⁸<http://www.redmine.org/>

Engine manages the workflow tasks that are created by the *Workflow Engine*. The Workflow Engine hosts and executes the process model.

Reflect Method

With this MESP task, the project team collects feedback about the method enactment in order to support the senior method engineer in improving method building blocks. The team might, for example, have feedback to improve the description of tasks or about the suitability of method services for their project.

In our scenario, Alice and her team are not satisfied with their standup meetings. As they are not so familiar with agile practices yet, they hope for more tips and hints in the task description. They use the execution logs of the BPEL engine and the work products they created to illustrate their issues with the method to the senior method engineer.

The MESP task *Reflect Method* is only indirectly supported with tooling. The BPEL engine tracks the execution of the method model, including the URIs for inputs and outputs. These can be found in the standard project repository. However, the actual findings are communicated informally to the senior method engineer.

Work Products and Tools

Figure 3.28 shows an UML class diagram that gives an overview of the relationship of the MESP work products discussed in this section. The BPEL Process model is executed within the BPEL engine that manages and logs Runtime Information about the state of execution. Whenever an HumanTask Invocation is executed, a Workflow Task is created and assigned to the person with the appropriate role.

Figure 3.29 shows an UML component diagram that gives an overview of the relationship of the MESP tools used in the MESP tasks that were discussed in this section. The Standard BPEL Engine that executes the process model consists of two components. The Workflow Engine manages the state of the process model and the Task Engine manages the state of workflow tasks. The Workflow Engine instructs the Task Engine to create workflow tasks. The Task Engine in turn notifies the Workflow Engine about finished workflow tasks. The Project Repository is used to store the created work products within the project. The location of work products is referenced within workflow tasks.

3.3 Summary

In this chapter, we provided an overview of the tasks, roles, work products, and tools that are part of the Method Engineering with Method Services and Method Patterns (MESP) solution. We first explained the tasks and associated roles to

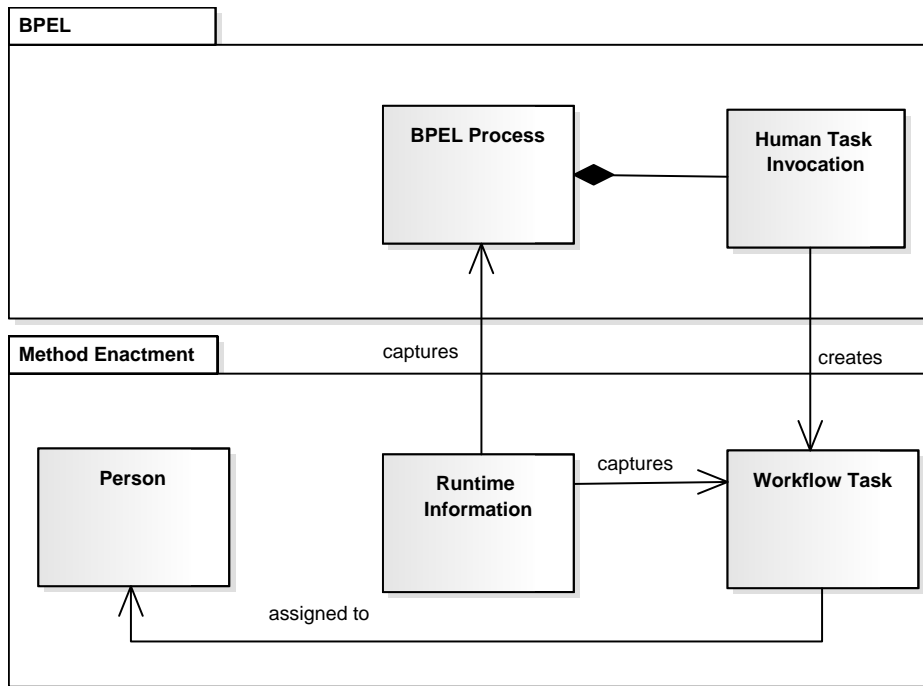


Fig. 3.28 Relationship of the MESP Work Products for Method Enactment

define method content, compose method models, and enact methods. We then presented the various work products that are created and used and discussed their relationship. We also explained what tools are used as part of our solution. Thereafter, we illustrated this with an end-to-end example from the eID domain. In the following chapters, we discuss the details of the method content definition, method tailoring, and method enactment with MESP.

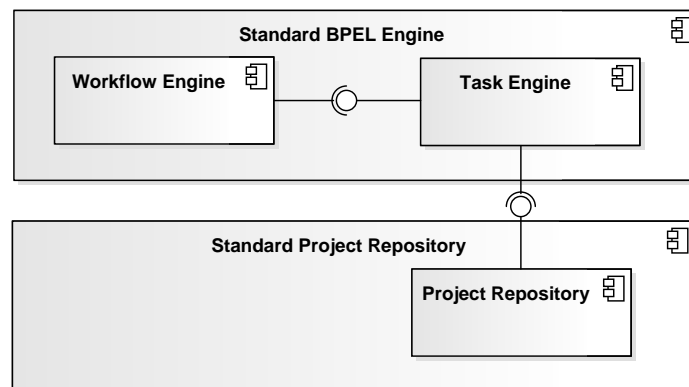


Fig. 3.29 The MESP tools for Method Enactment

CHAPTER 4

Method Content Definition

In the previous chapter, we presented the three roles of our solution and presented an end-to-end example. In this chapter, we discuss the details of method content definition by the senior method engineer.

This chapter is structured as follows. In Section 4.1, we first discuss requirements and related work. Thereafter, in Section 4.2, we explain two ways to extract reusable method content. In Section 4.3, we focus on the definition of basic elements. In Section 4.4, we discuss method services and method patterns that are based on these basic elements. Finally, we summarize the chapter in Section 4.5.

4.1 Requirements and Related Work

4.1.1 Requirements

4.1.2 Related Work

4.2 Extract Reusable Method Content

4.2.1 Extraction from Methods Described in Literature

4.2.2 Extraction from the Daily Practice of Organizations

4.3 Define Basic Elements

4.3.1 Definition of Basic Method Elements

4.3.2 Definition of Basic Characterization Elements

4.4 Define Method Services & Method Patterns

4.4.1 Definition of Method Services

4.4.2 Definition of Method Patterns

4.5 Summary

4.1 Requirements and Related Work

In this section, we describe the requirements and related work of method content definition. We first present the method definition requirements (method content definition requirements (MDRs)) that are a refinement of the solution requirements (SRs) presented in Section 2.2.1. Then we briefly summarize the related work that will be discussed also in the respective sections later.

4.1.1 Requirements

In this section, we discuss the requirements with respect to the method content definition for a holistic solution for software engineering method management based on an assembly-based method engineering approach.

In assembly-based method engineering, method building blocks need to be defined that can be reused later during method composition. Before these method building blocks can be defined, the actual method content need to be discovered and extracted. Thereby, method content can be found in methods described in literature like [AL12] and [SS13], or within the daily practice of organizations. Thus, the first task of a senior method engineer is to extract reusable method content from documentation or daily practice. The first MDR for our solution is therefore to *describe how to extract method content (MDR1)*.

Each method uses its own terminology, for example, the same work product might have different names in different methods. Thus, method content from different sources is inconsistent in terms of terminology. In order to create method building blocks that are interoperable and can be composed to method models, they need to be based on the same shared terminology. This base of terminology needs to be changeable to keep up with current trends, e.g., by introducing new concepts. Another MDR for our solution is therefore to *address the establishment of a common, updatable terminology for method building blocks (MDR2)*.

Method building blocks are composed based on the project characteristics later during method tailoring. In order to find suitable method building blocks, they shall be characterized as well. With only a few method building blocks, a small set of discriminating factors might be sufficient to pick the right ones. However, with a growing base of method building blocks, the characterization needs to be extensible such that additional situational factors and also situational factor values can be added. In order to support the method composition, a MDR is therefore to *address the establishment of a common, updatable characterization of method building blocks (MDR3)*.

In order to capture and formalize not only work products or tasks, but also more abstract method content, abstract orderings shall be supported by the solution (SR1.1). These abstract orderings shall be described using high-level modeling constructs (SR 1.5), so that senior method engineers without low-level language

experience are capable of defining them. We therefore formulate the MDR to *define a method building block type that allows modeling of abstract orderings with high-level modeling constructs (MDR4)*.

In order to compose method models with their flows of activities, method building blocks need to capture activities for the creation of software systems. In addition to simple activities, method building blocks need to be able to capture composite structures (SR1.3), so that complex structures of method building blocks can be abstracted to single method building blocks. Both need to be described using high-level modeling constructs (SR 1.5), so that senior method engineers without low-level language experience are capable of defining them. We therefore formulate the MDR to *define a method building block type that allows modeling both atomic activities and composite method building blocks with high-level modeling constructs (MDR5)*.

With solution requirement SR1.2, we described the need for interfaces of method building blocks. The more method building blocks are available, the more choices the project method engineer has, when composing a method model. However, the handling of these method building blocks becomes more difficult. In order to simplify the handling of method building blocks, interfaces need to abstract from the actual content and allow method engineers to compare interfaces instead of whole method building blocks. We therefore formulate the MDR to *define explicit interfaces for method building blocks that contain all the information necessary during method composition (MDR6)*.

With solution requirement SR1.4, we described the need for a method repository that stores the method building blocks for reuse by project method engineers. As method building blocks shall be added, deleted, and updated on the fly, it has to be ensured that the method repository stays consistent. In order to ensure the consistency among method building blocks, they shall be part of the same interconnected model, e.g., composite method building blocks shall reference and be consistent to the contained composite structures. Thus, we therefore formulate the MDR to *define a interconnected meta-model for method building block types and related information to support consistency (MDR7)*.

The discussed MDRs are summarized in Table 4.1. Also illustrated is the MESP task where the requirement needs to be addressed. In the following, we discuss each MESP task of our solution for method content definition. We then also explain how the respective requirements are met.

4.1.2 Related Work

Regarding the extraction of reusable method content, [Hen+14] provides an overview of the related work. While many works do not discuss how to actually extract method content, in [Ralo4] the author proposes two principal ways to define

Table 4.1 Method definition requirements and the affected MESP tasks

<i>Requirement</i>	Description	MESP Task
<i>MDR1</i>	describe how to extract method content	Extract Reusable Method Content
<i>MDR2</i>	address the establishment of a common, updatable terminology for method building blocks	Define Basic Elements
<i>MDR3</i>	address the establishment of a common, updatable characterization of method building blocks	Define Basic Elements
<i>MDR4</i>	define a method building block type that allows modeling of abstract orderings with high-level modeling constructs	Define Method Services & Method Patterns
<i>MDR5</i>	define a method building block type that allows modeling both atomic activities and composite method building blocks with high-level modeling constructs	Define Method Services & Method Patterns
<i>MDR6</i>	define explicit interfaces for method building blocks that contain all the information necessary during method composition	Define Method Services & Method Patterns
<i>MDR7</i>	define a interconnected meta-model for method building block types and related information to support consistency	Define Method Services & Method Patterns

method content for the creation of method building blocks. Other works illustrate how method content was extracted without discussing it explicitly [HLHo2]. In addition, more specialized literature on how to adopt a certain method may contain valuable insight, e.g., to assess the daily practice of software engineering [BRo4].

Regarding terminology-related basic method elements, several works discuss and present software engineering method meta-models, e.g., [ES10],[KFS13b], and [FHo2] that could be adopted to formalize basic method elements. With Software & Systems Process Engineering Meta-Model Version 2.0 [OMGo8], there exist a standardized meta-model that has gained attention from industry and research [KFS13b]. With respect to characterization of method building blocks many works discuss relevant situational factors for situational method engineering, e.g., [Bek+o8] and [CO12]. However, only few approaches consider a formalization and then often use key-value pairs, e.g., [Pli96],[Har97],[KDSo7].

Regarding the formalization of method building blocks, several works present approaches that define building blocks that differ in granularity, separation of

content and interface, and formality, e.g., [HB]94],[Bri96],[Cer+11],[RP96b],[RR01],[GLS98],[KW04],[Den+08].

Beside method building blocks that reflect work and work products directly, the notion of patterns was also proposed and discussed. While most works describe the usage of informal patterns [Cop95],[TR07] other approaches propose more formal pattern descriptions [DS98].

4.2 Extract Reusable Method Content

For the fulfillment of MDR₁, we want to discuss how reusable method content can be identified and extracted as a preliminary step towards the creation of method building blocks.

In order to populate the method repository with method building blocks, several authors propose to extract them from existing works, e.g., [RR01] or [Har97]. However, only very few explain how to do that [Ralo4]. In [HLHo2], the authors discuss a concrete example for the domain of web application development and explain, how they derived new method building blocks based on methods described in literature. In [Ralo4], the author proposes two principal ways to identify reusable method content for the creation of method building blocks (method chunks) and discusses two of them briefly. *Existing method re-engineering* deals with analysis and transformation of existing methods, while *ad-hoc construction* refers to the creation from scratch. In [Hen+14], the authors discuss the related work with respect to identifying and extracting method content, largely based on [Ralo4].

We summarize two approaches for *existing method re-engineering* [Ralo4] that we developed based on related work and our own experience that we believe to be valuable in practice. First, we discuss how to extract method content from methods described in literature, in part based on the results of a master thesis [Sie15]. Afterward, we describe how to extract method content from the daily practice of organizations as an example to extract undocumented method content. This approach is based on our experience in industry projects [FCE14; FGS15].

4.2.1 Extraction from Methods Described in Literature

Our approach to extract method content based on methods described in literature consists of three sequential preformed tasks as illustrated by Figure 4.1. In the following, we will discuss each task.

Identify Sources

In order to identify the suitable sources for method content extraction, we can differentiate two cases. In the first case, there is an explicit need for additional

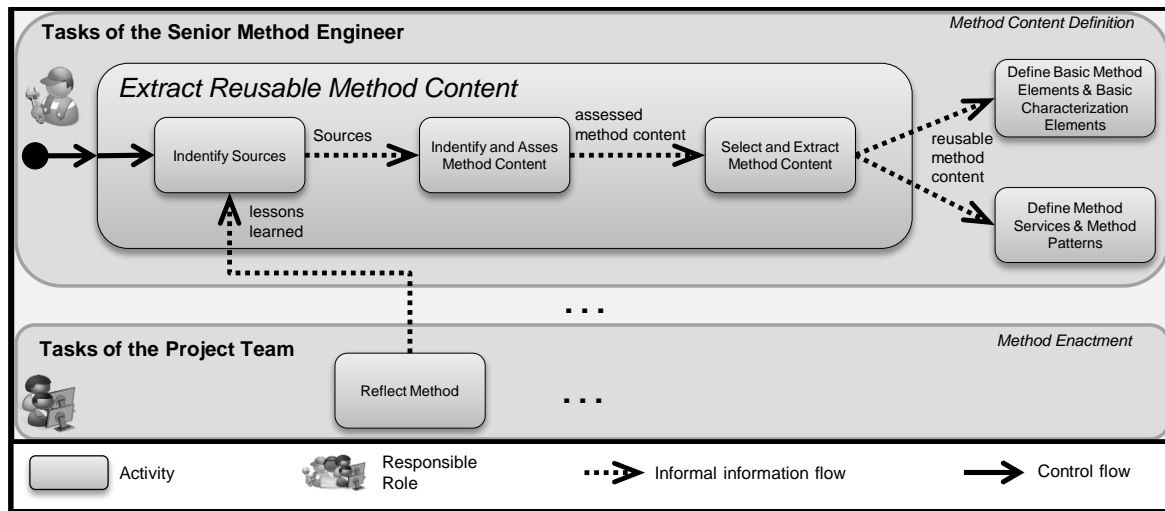


Fig. 4.1 Extraction of method content from methods described in literature

method content, e.g., based on lessons learned or missing method building blocks during method composition. In this case, the senior method engineer should identify and explicitly list all related problem areas and disciplines and then investigate them for related work as illustrated by [HLHo2]. Related work can be collected by querying scientific databases, e.g., SpringerLink⁹, IEEEExplore¹⁰, ACM Digital library¹¹, Google scholar¹² or SCOPUS¹³. In order to perform a more directed and formal procedure, relevant literature can be identified using a structured literature review [KC07] as illustrated by [Sie15].

In the second case, there is no concrete need that needs to be addressed; however, the method repository shall be updated with the latest trends and best practices. In this case, the newest proceedings of national and international conferences or issues of journals concerned with software engineering methods can be reviewed, e.g., PROFES¹⁴, ICGSE¹⁵, and ICSSP¹⁶ or TSE¹⁷, TOSEM¹⁸, and Journal of Software: Evolution and Process¹⁹. Searching for studies like [MW15],[KF15] that discuss the popularity and usage of software engineering methods is also a good approach.

⁹<http://link.springer.com>

¹⁰<http://ieeexplore.ieee.org>

¹¹<http://dl.acm.org>

¹²<http://scholar.google.com>

¹³<http://www.scopus.com>

¹⁴<http://www.profes-conferences.org>

¹⁵<http://www.icgse.org/>

¹⁶<http://www.icsp-conferences.org>

¹⁷<http://www.computer.org/web/tse>

¹⁸<http://tosem.acm.org>

¹⁹[http://onlinelibrary.wiley.com/journal/10.1002/\(ISSN\)2047-7481](http://onlinelibrary.wiley.com/journal/10.1002/(ISSN)2047-7481)

On the one hand, the identified work might present original work by the authors themselves that is worth disseminating directly. On the other hand, it might also discuss and reference other works, e.g., software engineering methods published as books [Engo8], industrial standards [ISO07], capability maturity models [Hoeo8], or organizational patterns [CHo5], which then serve as the source of method content that needs to be disseminated.

Identify and Asses Method Content

Especially, when many or extensive works are collected, it might be necessary to assess the contained method content to prioritize its dissemination. In the following, we explain a few heuristics that support the selection of suitable works.

Appropriateness Is the work related to the created list of problem areas and disciplines? Is it related to currently defined method building blocks? Work discussing methods that relate to these areas are probably more worthy to disseminate.

Granularity and Focus Is the work describing general-purpose methods or domain-specific methods? The first typically describe more generic tasks that are broadly applicable, but usually with less guidance and details. In addition, the terminology used is more generic. The latter typically provide specific descriptions and use a specialized terminology. Depending on the goals of the senior method engineer, content from general-purpose methods can fill gaps of missing method building blocks more easily, while content from domain-specific methods provides richer content for specific topics.

Formality Is the description of methods based on a meta-model? Is it very structured, e.g., by applying templates repeatedly or by using many sections? Typically, the more structured the method is described, the easier it is to understand and extract content to formalize it later. Informal and unstructured descriptions of methods tend to be ambiguous and more difficult to prepare for extraction. If a description of method is based on a meta-model that is similar to the MESP meta-model then only little effort is necessary to extract and reuse method content. However, if the meta-models are dissimilar, then the effort needed may become as high as with unstructured method descriptions [Hen+14].

Similarity Is the work using a similar terminology, that is, does it use tasks, roles, and work products that are the same or close to those that are already in the method repository? If this is the case, then it is probably easier to extract method content. However, the benefit of the additional method content might be smaller as it represents similar ideas.

Select and Extract Method Content

When the method content is not based on a meta-model or one that is different to our meta-model, it might become difficult to define basic element and method building blocks right away, because many methods were not created to be modular [Ralo4]. In addition, very extensive method descriptions are difficult to disseminate, if not done systematically. Thus, within this task method content is prepared for the definition based on our meta-model.

Based on the assessment described in the previous section, method content is selected for further dissemination. For each described method or its part, the following steps are applied. As a first step, hierarchical and chronological concepts are identified. Examples are *phases, stages, iterations, disciplines* or *decision gates*. Often, these concepts are used to structure the method and therefore are good candidates to systematically break up the method as illustrated by [FLE13]. These orderings are also potential candidates to be later formalized with method patterns.

As a second step, each of the identified concepts are decomposed further into groups of interconnected concepts. Examples for what we are looking for now are *tasks, activities, task descriptions, events, roles, work products, states* and *artifacts*. Important connections are for example *containment, predecessor, successor, input of, output of, responsible for*. The basic method elements used for these concepts and their relationships might be mentioned explicitly or only implicitly. In the latter case, the senior method engineer needs to identify them and make them explicit.

In order to capture and document both kinds of concepts, based on our experience, natural language and semi-formal diagrams should be used, as it is often to difficult to formalize the concepts directly. However, if based on a similar meta-model, concepts can already be formalized by instantiation of basic method elements from our meta-model.

As a last step, the terminology and situational factors mentioned in the method content are investigated. In order to define interoperable method building blocks, they must share the same base of basic method elements. As each method uses its own terminology, e.g., with respect to role or work products names, its terminology needs to be mapped to the one used in the method repository. This can be done by creating a mapping from the terms used in the method content to the names of basic method elements in the method repository. The mapping can be provided with a textual table. In order to define interoperable method building blocks, they must also share the same base of situational factors. In order to collect suitable situational factors, potential situational factors mentioned within the method descriptions should be listed in a textual table for later use. The senior method engineer should also reference identified concepts that are related to these situational factors according to the method description.

After these steps, the senior method engineer has extracted reusable method content from methods described in literature. She can then formalize it by creating basic elements, method services, and method patterns.

4.2.2 Extraction from the Daily Practice of Organizations

Our approach to extract method content based on the daily practice of organizations consists of five tasks as illustrated by Figure 4.1. The first two task are enacted in parallel and lead to the task for the creation of method requirements. These first three tasks serve as the foundation of the following two tasks *Process Enactment Feedback* and *Select and Extract Method Content*. These two tasks are periodically carried out in order to continuously extend and refine the method repository based on the experience gained. In the following, we will discuss each task.

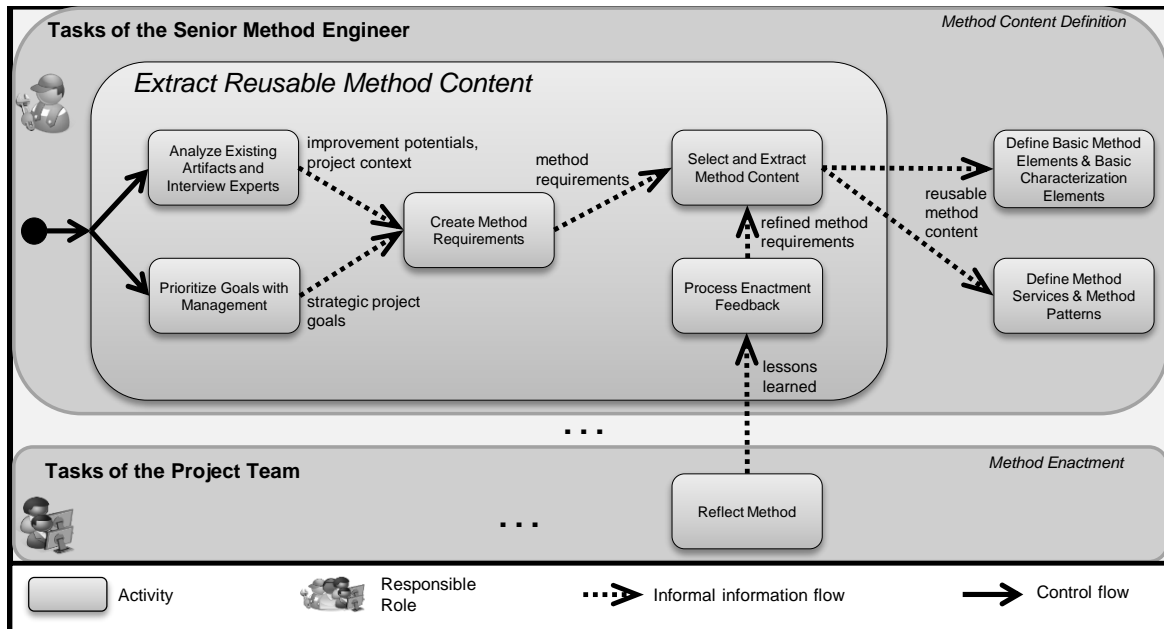


Fig. 4.2 Extraction of method content from the daily practice of organizations

Analyze Existing Artifacts and Interview Experts

Before the senior method engineer can extract reusable method content from the daily practice of an organization, she needs to establish an overview of how software systems are created within that organization.

Based on our experience [FCE14] and as proposed in [BR04], we propose to conduct interviews with project teams in order to capture the enacted method and its strengths and weaknesses. Relying solely on the documentation of methods results in less effort. However, there is the risk that it might not reflect the daily

practice, when the method is typically not enacted according to its definition and both deviate from each other. In our opinion, interviews are a good trade-off between the effort for the analysis and the quality of the results. If desired, other approaches can be used, e.g., collaborative workshops [FR15] or apprenticing [AW05] that might cause more effort, but also give different results.

In order to prepare the interviews, we propose to analyze existing artifacts related to the software engineering method. These are work products created within software projects, e.g., *requirement specifications* or *design documents*, but also *method descriptions* that describe the method or parts of it, e.g., *task descriptions*, *document templates*, or *checklists*. Based on the analysis, the senior method engineer can already postulate characteristics of the method, possible weaknesses, and strengths. One weakness might be that documentation for the implementation is incomplete for some projects.

With the results of the analysis, the senior method engineer can then prepare the interviews with the project teams. She prepares questions in order to back up or refute the analysis results. The interviews should be representative such that with the involved project team members all roles, tasks, work products, and most situations should be covered. We propose to have the interviews with two interviewers and two interviewees as this provides second opinions when discussing the questions (interviewees) and later the results (interviewers). The questions should cover weaknesses and strengths regarding all parts of the software engineering method: activities, work products, roles, tools, the process, and the flow of information. They should also cover the method description with its provided guidances. Another important aspect are situational factors. The questions should assess, e.g., what situational factors are differentiated explicitly (e.g. local and global software projects) and what situational factors have influence on the used method, e.g., by affecting the performed tasks or used tools. One result might be that in the past global software projects, often project team members had differing expectations regarding the required documentation of the implementation.

The results of artifacts analysis and expert interviews are used to derive method requirements and to extract method content in the following tasks.

Prioritize Goals with Management

A software engineering method is used by a project team to create a software system in a software project. While the project team has the primary goal of creating that software system within time and budget, the management of the organization pursues middle and long term goals. These might require mandatory activities within a method that are not primarily beneficial for that same project, but of strategically interest of the organization. One example might be the long-term maintainability of the software systems.

In order to identify and consider these strategic method improvement goals, the senior method engineer should carry out workshops with the stakeholders that are responsible for the software projects. She can propose strategic method improvement goals based on the results of the interviews to initiate the discussion. Once a set of goals is identified, goals have to be prioritized with the managers, e.g., by letting each of them distribute a smaller amount of priority points among the goals.

The prioritized strategic method improvement goals are used to derive method requirements in the following task.

Create Method Requirements

Especially, when not all aspects of the software engineering methods can be addressed at once, it is necessary to prioritize among them. Similar to [RR01], we therefore propose the use of method requirements to make the goals of method content extraction more transparent and to prioritize them with the stakeholders. Method requirements describe what needs to be ensured by methods, e.g., that the project team discusses the mutual expectations at project start or that certain tools are used. Thereby, method requirements provide guidance on what method content to extract (first).

The senior method engineer proposes method requirements based on both, the results of the expert interviews and the prioritized strategic method improvement goals. As with the strategic method improvement goals, she then should carry out workshops with the stakeholders that are responsible for the software projects and discuss the method requirements. Once a set of method requirements is identified, it has to be prioritized by the managers, e.g., by letting them each distribute a smaller amount of priority points among the requirements.

The prioritized method requirements are used as the basis to select and extract method content in the following task.

Select and Extract Method Content

With this task, method content is prepared to be formalized within the following tasks. The goal is to iteratively select and extract method content from the daily practice within the organization based on the initial set of method requirements and the refinements from the method enactment. Weaknesses in the existing methods are addressed by additionally disseminating method content from methods described in literature.

Based on the prioritized method requirements, the senior method engineer selects and extracts method content from the daily practice of the organization. Similar to the initial analysis she can investigate artifacts and interview project teams to do this. The goal is to capture concepts that are perceived as being successful

and proven by the project teams and that address the method requirements. If method requirements cannot be addressed by the daily practice of the project teams, the methods described in literature are investigated (see Section 4.2.1) for solutions. As before, the senior method engineer documents tasks, task descriptions, the process, roles, work products, guidances (tools, templates, checklists), and associated situational factors. For example, the method requirement for the project team to discuss the mutual expectations at project start could be fulfilled by a mandatory kick-off workshop, where these are discussed and documented. Based on the experience, this might be mandatory only for global software projects.

Based on our experience, natural language and semi-formal diagrams should be used to capture the extracted method content and reference existing method descriptions as formalizing the concepts directly is often too difficult. By using an online document that is accessible via browsers, stakeholders can easily feedback the extracted method content and the senior method engineer can iteratively refine it. If the organization already uses a method model based on a meta-model similar to our meta-model, concepts can already be formalized by instantiation of basic method elements from our meta-model.

Once feedback from method enactment is available it is used to refine the method requirements within the task *Process Enactment Feedback*. Similar to the initial set of method requirements these are used to select and extract method content from the daily practice of the organization and, if required, method requirements.

As described in Section 4.2.1, the terminology and the situational factors need to be investigated and prepared, e.g., using textual tables. However, when extracting method content from the daily practice within the same organization typically it is more homogenous and does not require special preparation.

After this step, the senior method engineer has extracted reusable method content from the daily practice of organizations. She can then formalize it by creating basic elements, method services, and method patterns.

Process Enactment Feedback

With this task, the senior method engineer uses the feedback from method enactment to check the fulfillment of method requirements and to possibly refine them. This ensures that the defined method building blocks fulfill the actual requirements and helps to maintain the transparency for the stakeholders of software projects.

Based on the lessons learned of the project teams, the senior method engineer checks whether the method requirements were fulfilled and whether additional method requirements are necessary. She marks method requirements as fulfilled, e.g., when former weaknesses are successfully mitigated. She adds additional and refined method requirements, if the lessons learned indicate that this is necessary. One example might be that although mutual expectations are discussed and documented at project start, the documentation for the implementation is still

incomplete for some projects. Then a refined method requirement might be to review the documentation of the implementation at the end of an iteration.

After this step, the senior method engineer starts another iteration of method content extraction with the task *Select and Extract Method Content*.

4.3 Define Basic Elements

In this section, we discuss how our solution addresses the establishment of a common, updatable terminology (MDR₂) and characterization (MDR₃) of method building blocks. To fulfill these requirements, we need to define meta-classes that can be used to formalize the available terminology and characterization. These basic elements serve as a common frame of reference as all defined method building blocks will be based on them.

With respect to terminology-related basic method elements, several works discuss and present software engineering method meta-models. In [ES10], the authors present the meta-model of their creation-based situational method engineering approach MetaME and relate its meta-classes to the meta-classes of other popular approaches. In [Spi15], MetaME is extended to support continuous software method improvement. One of the first comprehensive software engineering method meta-models is OPEN [FH02]. Another comprehensive meta-model is provided by the software engineering method framework V-Modell XT that is prescribed for projects with the German government [KTF11], but a purely German standard with little international attention [KFS13b]. Standardization efforts resulted in two standards. First, the Software Engineering Metamodel for Development Methodologies known as the ISO/IEC 24744 International Standard [ISO07] and, second, Software & Systems Process Engineering Meta-Model Version 2.0 [OMGo8]. While the first has gained no impact and practical relevance, the latter has gained attention from industry and research [KFS13b].

With respect to characterization of method building blocks, many works discuss relevant situational factors for situational method engineering, however only few approaches consider a formalization. In [Bek+08], the authors propose a set of 27 situational factors identified as relevant for software product management. In [CO12] the authors consolidate related research into an initial reference framework of 44 situational factors and additional sub-factors affecting situational method engineering. Situational method engineering approaches often use fixed sets of key-value pairs to formalize situational factors [Pli96],[Har97],[KDS07] or do omit an explicit characterization [RDR],[KLR96],[Ell+11].

In the following, we explain the basic elements of our software engineering method meta-model. Following the structure of our software engineering method meta-model illustrated in Figure 4.3, we first discuss basic method elements and

show, how they are used to formalize terminology. Thereafter, we discuss basic characterization elements used to model situational aspects.

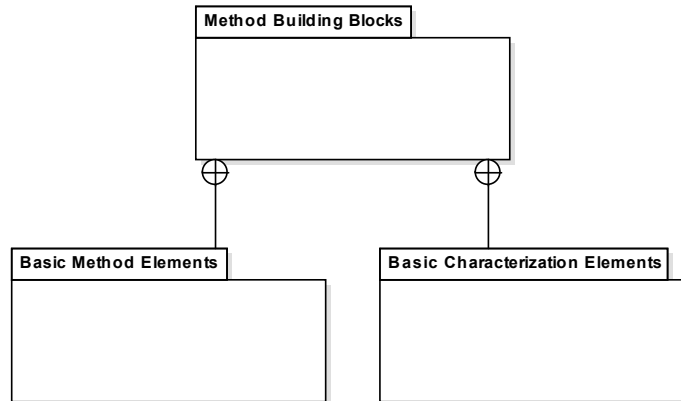


Fig. 4.3 Excerpt of package structure of MESP meta-model

4.3.1 Definition of Basic Method Elements

The basic method elements of our software engineering method meta-model are defined in reference to the tool implementation of SPEM, the EPF Composer²⁰, for the following three reasons. First, SPEM is the most widespread software engineering method meta-model and it allows defining and updating method related terminology (MDR2). Second, with the EPF Composer there is tool support and an implementation of the meta-model for the popular development platform Eclipse available. Third, a large library of method content formalized with the EPF Composer is publicly available.

As illustrated by Figure 4.4, SPEM separates reusable method content from its application in processes (methods). Thus, it differentiates between *method content* meta-classes and *process* meta-classes. Method content describes, e.g., tasks, roles, and work products that can be used in potentially many different processes, which reference this method content. Guidances can be defined with respect to both, method content and processes. As the process part is not relevant to our basic method elements, in the following, we focus on the method content of SPEM.

As illustrated in Figure 4.4, to define method content in SPEM the meta-classes Task Definition, Step, Work Product Definition, Role Definition, and Guidance are used. Additionally and not shown in the figure, Task Definitions can associate a list of Qualifications instances that documents the qualifications required for the performance of a task. In addition, Task Definitions can associate a list of Tool Definitions that can be used to support the task.

²⁰<http://eclipse.org/epf/>

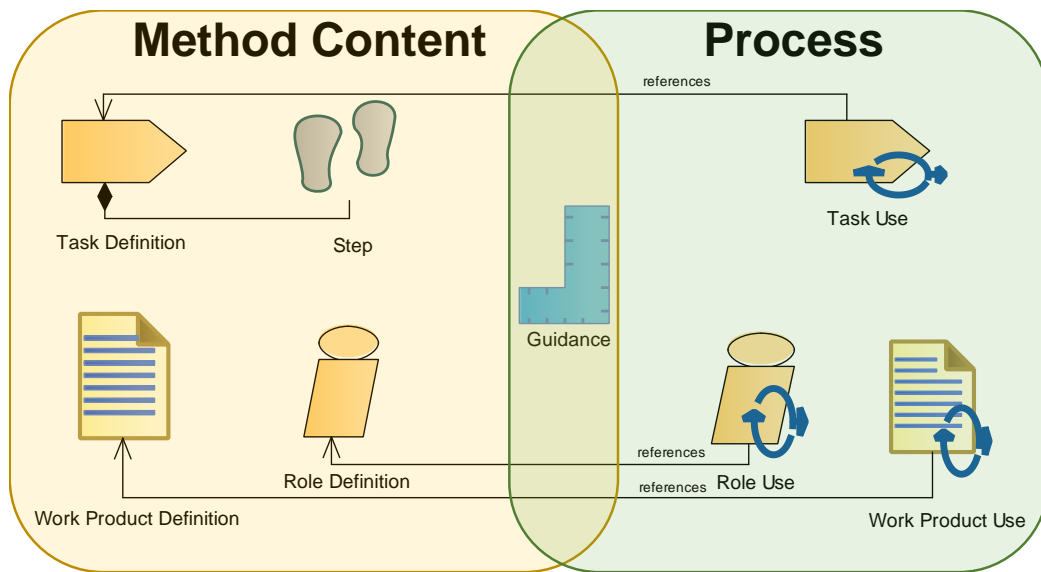


Fig. 4.4 Illustration of division between Method Content and Process in SPEM (adopted from [OMG08])

For our solution, we adopted the method content meta-classes of SPEM's implementation, the EPF Composer. Thus, details slightly deviate from the SPEM standard. As we adopted only parts of the meta-model, we also simplified the structure of the meta-model by omitting abstract meta-classes and moving their relationships and attributes to the inheriting meta-classes.

Meta-Classes

In this section, we discuss the basic method element meta-classes of our MESP meta-model that are illustrated in Figure 4.5 and were adopted from SPEM.

Task Task is based on the SPEM Task Definition meta-class. It represents work being performed by Roles and it is associated with input and output Work Products. Inputs are differentiated in mandatory versus optional inputs. A Task must have a single Role that is responsible for it during method enactment and can have additional Roles that are associated as additional performers. A Task can recommend a specific set of Tools to be used to support the Task. It can also provide a list of Qualifications that the task typically requires to be performed versus optional inputs. A Task must have a single Role that is responsible for it during method enactment and can have additional Roles that are associated as additional performers. A Task can recommend a specific set of Tools to be used to support the Task. It can also provide a list of Qualifications that the task typically requires to be performed by one or more Roles and this list can be mapped against the provided Qualification list defined for Role. As illustrated,

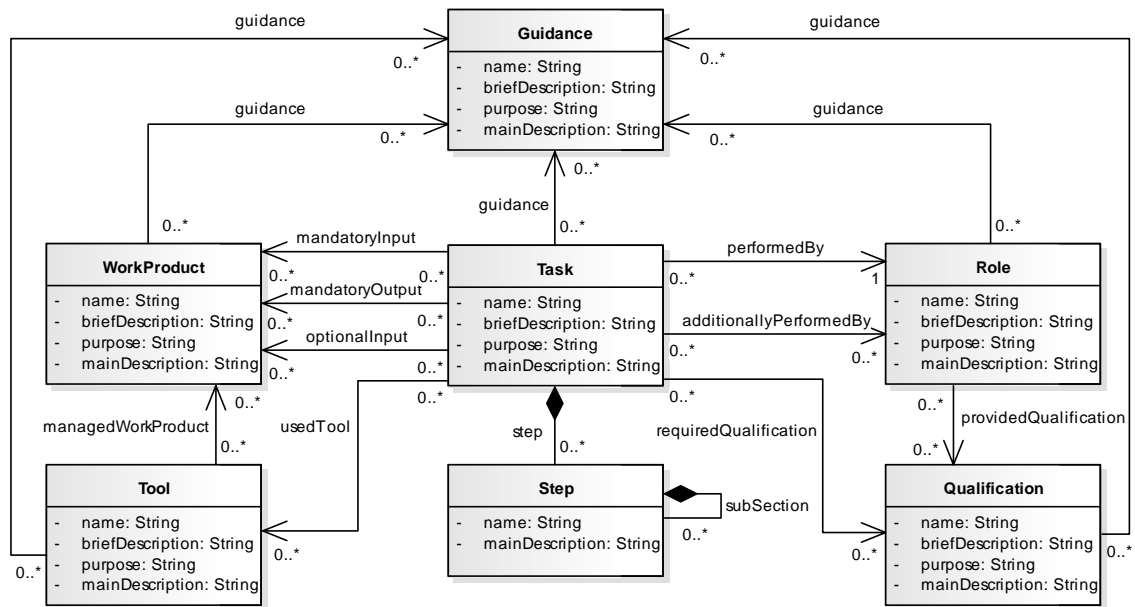


Fig. 4.5 The basic method elements of our meta-model package BasicMethodElements

Task contains several attributes that allow giving it a name, its description, and to describe its purpose.

Step Step is based on the SPEM Step meta-class. It is used to organize a Task's description into parts or subunits of work. Steps can describe sub-steps nested as subSections into Steps. As illustrated, Step contains attributes that allow giving it a name and a description.

Role Role is based on the SPEM Role Definition meta-class. It represents a set of related skills, competencies, and responsibilities and can provide a list of Qualifications. It is used by Task to define who performs it. As illustrated, Role contains several attributes that allow giving it a name, its description, and to describe its purpose.

Tool Tool is based on the SPEM Tool Definition meta-class. It represents a tool that can be used to process a WorkProduct and it is used by Task to define its participation in performing the Task. As illustrated, Tool contains several attributes that allow giving it a name, its description, and to describe its purpose.

Qualification Qualification is based on the SPEM Qualification meta-class. Qualification documents required qualifications, skills, or competencies for Roles and/or Tasks. Qualifications can be used to find suitable project team

members to assign them to a Role for method enactment. As illustrated, Qualification contains several attributes that allow giving it a name, its description, and to describe its purpose.

Guidance Guidance is based on the SPEM Guidance meta-class. Guidance provides additional information related to the associated element, e.g., guidelines, templates, checklists, or examples. As illustrated, Guidance contains several attributes that allow giving it a name, its description, and to describe its purpose.

Usage

By adopting the standard SPEM, we cover all types of basic method elements that are relevant. With these, the terminology used in method models can be extended and updated as required by senior method engineers. For example, let us consider that the senior method engineer extracted method content from the OpenUP method as illustrated in our end-to-end example in Section 3.2. OpenUP describes the usage of an *architecture notebook* “to capture and make architectural decisions and to explain those decisions to developers”²¹. If the notion of architecture notebook did not exist in her method repository yet, the senior method engineer would not be able to describe method services that have architecture notebook as input or output. However, she can introduce the concept of architecture notebook by defining a new WorkProduct instance as illustrated in Figure 3.10. Senior and project method engineers are then able to use it in method building blocks and method models. To the contrary, this would not be possible, if the set of work products was predefined by us.

4.3.2 Definition of Basic Characterization Elements

The basic characterization elements of our software engineering method meta-model are defined with respect to the prevalent formalization of situational factors that is the usage of key-value pairs. As illustrated by the end-to-end example in Section 3.2, beside situational factors we also need to characterize method building blocks with categories. Situational factors help to characterize in terms of project situation, while categories help to characterize method building blocks in terms of, e.g., origin, discipline, or typical phase. Basic characterization elements need to be extensible, similar to basic method elements. The senior method engineer needs to be able to add new situational factors and categories (e.g. `customer_involvement`, `phase`) or respective values (e.g. `customer_involvement.high`, `phase.design`).

²¹http://epf.eclipse.org/wikis/openup/practice.tech.evolutionary_arch.base/workproducts/architecture_notebook_9BB92433.html

Meta-Classes

In this section, we discuss the basic characterization element meta-classes of our MESP meta-model as illustrated in Figure 4.6.

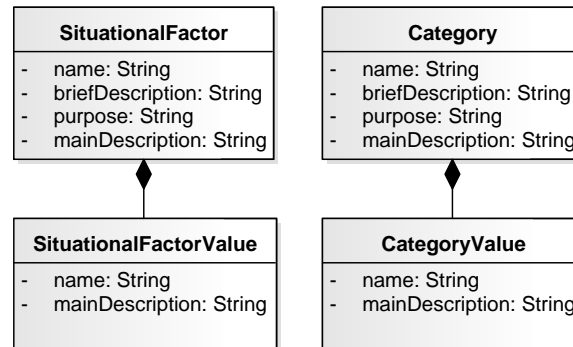


Fig. 4.6 The basic characterization elements of our meta-model package BasicCharacterizationElements

SituationalFactor SituationalFactor represents a characteristic of the project or organization that influences the suitability of method building blocks for a method model. Thus, SituationalFactors span the domain of definition to describe a situation. A SituationalFactor contains SituationalFactorValues that represent its co-domain. As illustrated, SituationalFactor contains several attributes that allow giving it a name, its description, and to describe its purpose.

SituationalFactorValue SituationalFactorValue represents a possible value of a specific SituationalFactor. Thus, all the SituationalFactorValues of a SituationalFactor span its co-domain. As illustrated, SituationalFactorValues contains attributes that allow giving it a name and a description.

Category Category represents a characteristic that is not a SituationalFactor, but relevant to discriminate method building blocks. Categories span the domain of definition to classify them and contains CategoryValues that represent its co-domain. As illustrated, Category contains several attributes that allow giving it a name, its description, and to describe its purpose.

CategoryValue CategoryValue represents a possible value of a specific Category. Thus, all the CategoryValues of a Category span its co-domain. As illustrated, CategoryValues contains attributes that allow giving it a name and a description.

Usage

With our formalization of characterization elements, the characterization of method building blocks and projects can be extended and updated as required by senior method engineers. For example, the senior method engineer can add a new situational factor and respective values for customer involvement as illustrated in the end-to-end example in Section 3.2. As another example, let us consider that the senior method engineer extracted method content from the plan-driven methods based on V-Modell XT and OpenUP before. Now she extracts method content from the agile method Scrum and wants to be able to discriminate plan-driven and agile method building blocks. The notion of plan-driven or agile was not existent before, so method building blocks could not be differentiated by their *nature*. However, she can introduce a new Category instance as illustrated in Figure 3.10. She would then be able to characterize the method building blocks accordingly. To the contrary, the senior method engineer would not be able to extend the characterization elements, if they were predefined by us.

4.4 Define Method Services & Method Patterns

In this section, we discuss the method building block types provided by our solution, method services and method patterns. We describe how we enable the modeling of both atomic and composite activities with our method services (MDR5) and how we enable the modeling of abstract ordering with our method patterns (MDR4). We illustrate their interfaces that offer the information necessary during method composition (MDR6) and explain how the meta-model classes of method services and method patterns relate to basic method elements to support consistency (MDR7).

Regarding method building blocks for software engineering method models, various approaches are discussed in literature. Several authors adopted the terms *method fragments*, *method chunks*, *method components*, and *method services*.

The term method fragment was defined in [HBJ94] and became popular by its use in [Bri96]. In practice, method fragments are usually thought of as the smallest building blocks of a method, typically defined in terms of an element in a meta-model [HV97],[Hen+14]. Typically, there are at least two types of method fragments, process fragments and product fragments, with some kind of association between them, as for example in [Cer+11].

The term method chunk relates to the combination of a process part and a product part [RP96b],[RRo1],[Ralo4],[MRo6] and thus a concept that is more coarse-grained. As method chunks link a process fragment to many product fragments, the number of elements to create a method model is decreased. In addition, method chunks possess interfaces to describe the situations where they can be applied

meaningfully. However, in contrast to the information hiding rationale used in object-oriented programming, both, fragments and interface, need to be visible to the method engineer [Hen+14].

The term method component promotes the view of methods as constituted by exchangeable and reusable components [GLS98]. In [KW04] and [KW06], it is defined as a *self-contained part of a method, expressing the transformation of one or several artifacts into a defined target artifact and the rationale for such a transformation*. Similar to the interface of method chunks, the method rationale capture the reasons why a particular method component is useful in a particular context and why it is designed the way it is. In addition, it allows hiding unnecessary details during method tailoring [KÅ09]. However, method components were proposed for a configuration-based situational method engineering approach.

The term method service was coined in [Den+08]. Based on service-oriented principles, a method service comprises two basic parts: first, a descriptor part with a semantic description of the purpose and operationalization of the method service and, second, an implementation part that contains the executable description of the method service. On top of the method service idea, Rolland has sketched concepts of *method as a service* and *method-oriented architecture* according to the service-oriented approach [Rolo9]. In [Cau10], the authors present another service-oriented approach called SO2M. In contrast to method components, method services were proposed for assembly-based situational method engineering approaches, however, there is still lack for a holistic tool-supported approach.

Regarding the definition of abstract orderings, in [MR06], the authors discuss beside method chunks and method fragments the notion of *patterns* and *road-maps*.

The notion of patterns is commonly used to describe *a general solution to a common problem or issue, one from which a specific solution may be derived* [Cop95],[Amb98]. Similarly, a process patterns describes an pattern of activity that has shown to be successful in practice and that was derived by abstraction from recurring software development methods [Cop95],[TR07]. For example, in [TR07] the authors provide a set of high-level process patterns for agile development that they derived from several agile methods. In [GJS10], the authors describe a procedure to extract process patterns out of existing software engineering methods. However, these patterns are typically described informally. In [PM04], the authors use diagrams based on SPEM to illustrate their patterns that are still mainly described in natural language. In [RP96b] and [RP96a], the authors propose generic method construction patterns formalized with method chunks. These reflect common patterns of behavior that a method engineer should use to tailor methods. Thus, they do not reflect method building blocks, but are part of the meta-method to create methods. In [DS98], the authors discuss a different notion of patterns that *encapsulate knowledge about processes that can be reused and applied in different settings*. Thus, this notion of patterns is close to the original pattern idea. Here, a pattern consists of an interface that

describes in which situation the pattern is applicable and a body that uses a method chunk to describe the pattern.

With respect to road-maps, in [MRo6], the authors discuss an approach that uses road-maps additionally to method chunks. A selected set of method chunks does not necessarily predetermine the sequence in which the method chunks have to be used. That means, that different road-maps among a set of method chunks are possible, where a road-map represents a path in a method model or a specific sequence of method chunks in a method model. The idea described in the paper is to add a second step of road-map-driven method configuration to find the road-map, which is the most suitable for the needs of a team member.

Based on the requirements and the related work, we define two types of method building blocks in our software engineering method meta-model that are built on top of the basic elements. Following the structure of our meta-model illustrated in Figure 4.7, we first explain how method services are formalized. Thereafter, we explain the formalization of method patterns.

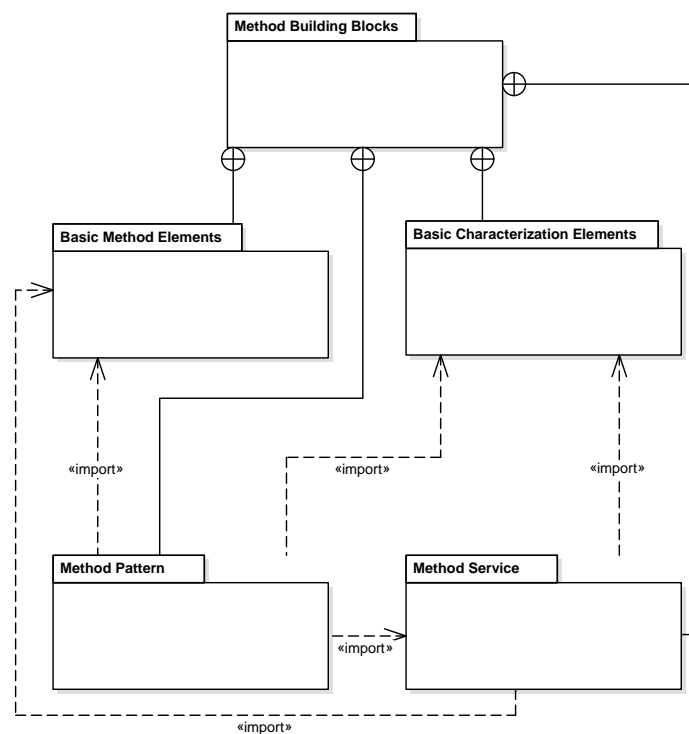


Fig. 4.7 Excerpt of the package structure for method building blocks of the MESP meta-model

4.4.1 Definition of Method Services

The main type of method building block in our solution is the method service that follows the service-oriented paradigm as described by [Den+08], [Rolo9], and

[Cau10]. However, beside the conceptualization, we require a formalization and executable notion of method services for our solution for software engineering method management. Especially, our method services have to fulfill the stated requirements MDR5, MDR6, and MDR7.

In order to allow for the definition of executable method services that reflect both atomic and composite activities (MDR5), we use a variation of the composite pattern [Gam+07] in our meta-model and embed a process within each method service. This process is then comprised of a single reference to a task (atomic activity) or a flow of method service references (composite activity). Following the principles of other approaches based on method services, our method service comprises two parts, interfaces and a process. The interfaces describe the information necessary during method composition (MDR6), while the process part contains the operationalization of the method service. Both, the interfaces and the process part are designed to reuse basic elements instead of duplicating them in order to support consistency (MDR7).

Meta-Classes

In this section, we discuss the method service related meta-classes of our MESP meta-model as illustrated in Figure 4.8.

MethodService `MethodService` is the main method building block of our approach. It represents the composable notion of work, both an atomic task or a flow of composed method services. `MethodService` contains two interfaces to describe its suitability with respect to situations and other method services. It also contains an executable `Process` with the actual description of work – the tasks that need to be performed. This process is the same meta-class that is used within method models (see Section 5.3.2). It references method services and tasks via `TaskDescriptors` and `MethodServiceDescriptors` to denote that they are executed as part of the process and thus, as part of the method service (control and data flow meta-classes used in a process are omitted in Figure 4.8). As illustrated, `MethodService` contains several attributes that allow giving it a name, its description, and to describe its purpose.

CharacterizationInterface `CharacterizationInterface` characterizes the suitability with respect to a situation by referencing suitable and unsuitable situational factor values. In addition, it references suitable category values.

StructuralInterface `StructuralInterface` characterizes the technical interoperability with respect to work products and roles. It describes the work products that are optionally or mandatory required in order to work properly and the work

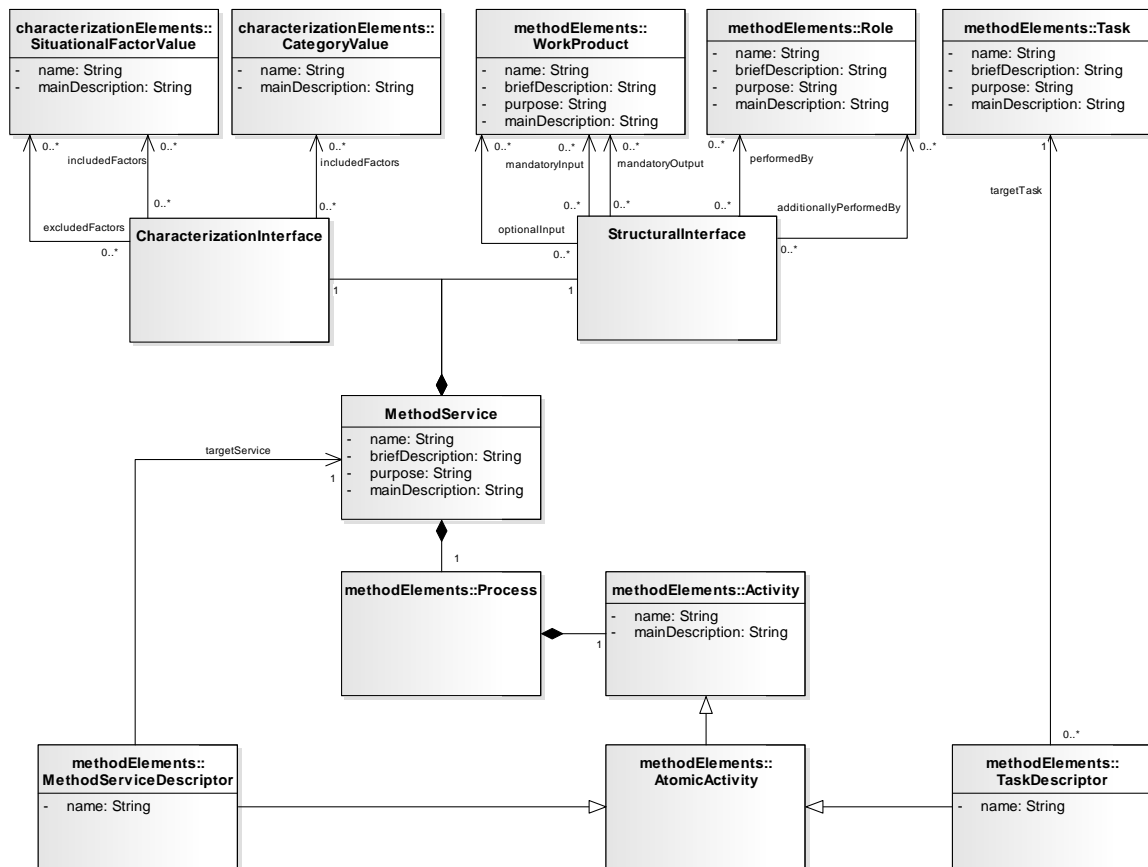


Fig. 4.8 The method service related meta-classes of our meta-model package MethodService

products that are produced. It also describes the roles that are used within the characterized unit, e.g., a method service.

Usage

With our formalization of method services, we enable the modeling of atomic and composite activities as the process of a method service can reference a single task or a complex flow of method services. Thus, the senior method engineer can summarize several tasks to more coarse-grained, executable, and composable units of work. Figure 4.9 shows a refined illustration of the method service derived from OpenUP as shown in 3.11. As illustrated, both parts of the method service, the interface part and the process part, reuse basic elements to express the usage of work products, roles, and tasks as well as situational factor values and category values.

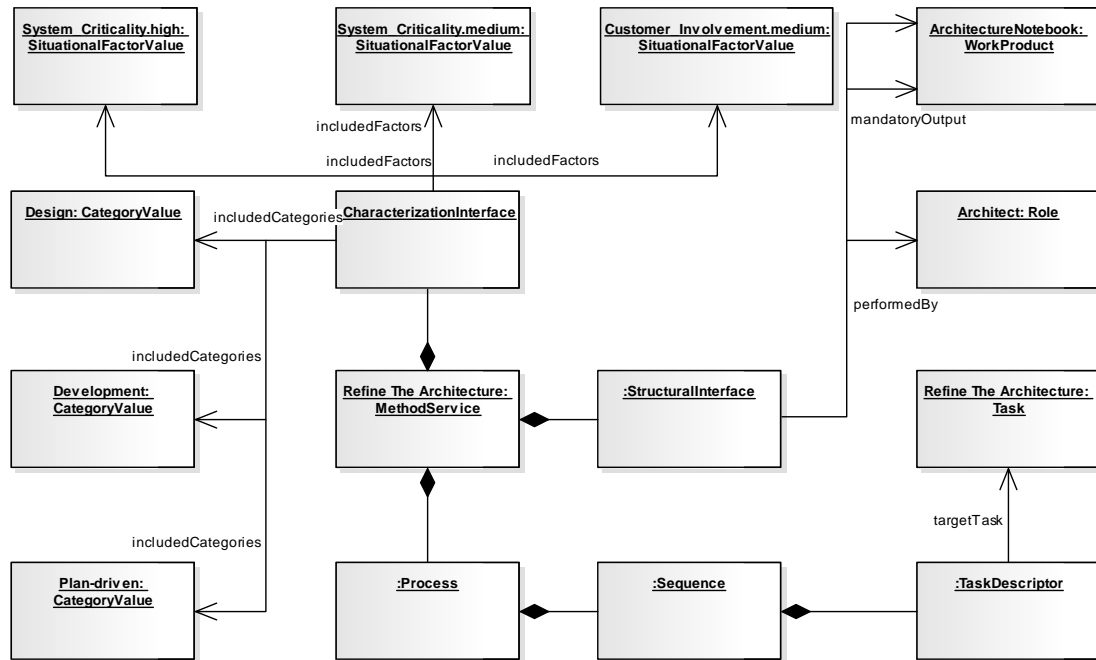


Fig. 4.9 A method service derived from OpenUP

4.4.2 Definition of Method Patterns

The second type of method building block in our solution is the method pattern that follows the pattern-based ideas as described by [TR07]. Our method patterns need to describe abstract orderings (MDR₄) and thus abstract from concrete method services. However, while patterns are typically described informally, we need sufficient formalization of these patterns in order to help project method engineers to model executable method models and to precisely evaluate their fulfillment during the quality assurance of method tailoring. Thus, our notion of patterns goes beyond formalizations as presented in [DS98]. Our method patterns contain scopes that are connected by executable control flow and where each scope contains a constraint. A method pattern is successfully implemented in a method model, when all constraints are fulfilled. Constraints are defined with a Domain Specific Language (DSL), such that senior method engineers does not have to use low-level modeling, while constraints can still be evaluated automatically. In addition, our method services have to fulfill the stated requirements MDR₆ and MDR₇.

Meta-Classes

In this section, we discuss the method pattern related meta-classes of our MESP meta-model. The main meta-classes are illustrated in Figure 4.10. The abstract syntax of the DSL to model constraints is also defined within the MESP meta-model.

The related meta-classes are illustrated in Figure 4.11, Figure 4.12, and Figure 4.13 and will also be described in the following.

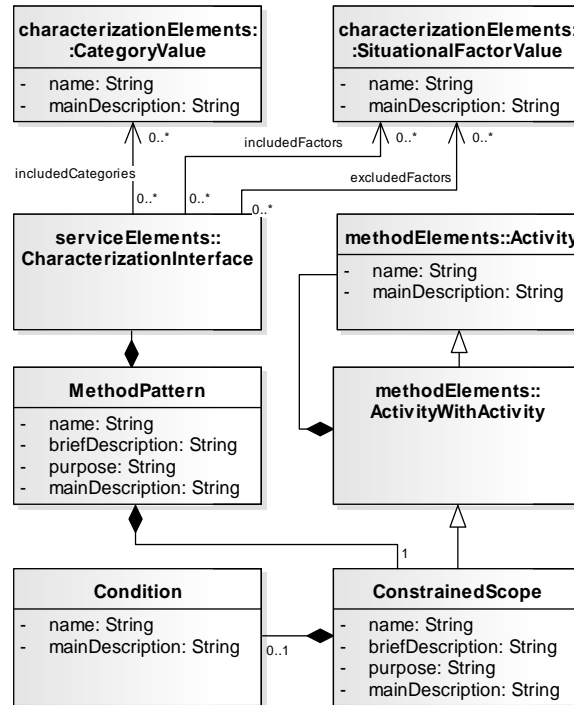


Fig. 4.10 The method pattern related meta-classes of our meta-model package MethodPattern

MethodPattern MethodPattern is the second type of method building block of our approach. It represents abstract orderings of groups of method services, where these groups are paraphrased by constraints that need to be fulfilled by using the right method services. A MethodPattern contains a ConstrainedScope that represents the actual pattern content and a CharacterizationInterface that characterizes the suitability of the method pattern (see Section 4.4.1).

As illustrated, MethodPattern contains several attributes that allow giving it a name, its description, and to describe its purpose. In particular, the *mainDescription* attribute can be used to describe intent, problem, context, and forces with natural language, as known from pattern descriptions like in [CS95].

ConstrainedScope ConstrainedScope represents the content of a method pattern. It consists of its content and a Condition that this content needs to fulfill. A control flow either can host an empty control flow element like a sequence or further constrained scopes connected by control flow. Thus, a ConstrainedScope can host a complex structure of constrained scopes.

As illustrated, ConstrainedScope contains several attributes that allow giving it a name, its description, and to describe its purpose.

Condition Condition formalizes the constraint of a ConstrainedScope that needs to be fulfilled by the method service descriptors used within a Constrained-Scope. The constraint is expressed with the DSL created for the usage by senior method engineers illustrated in the Figures 4.11–4.13.

As illustrated, Condition contains attributes to give it a name and a description.

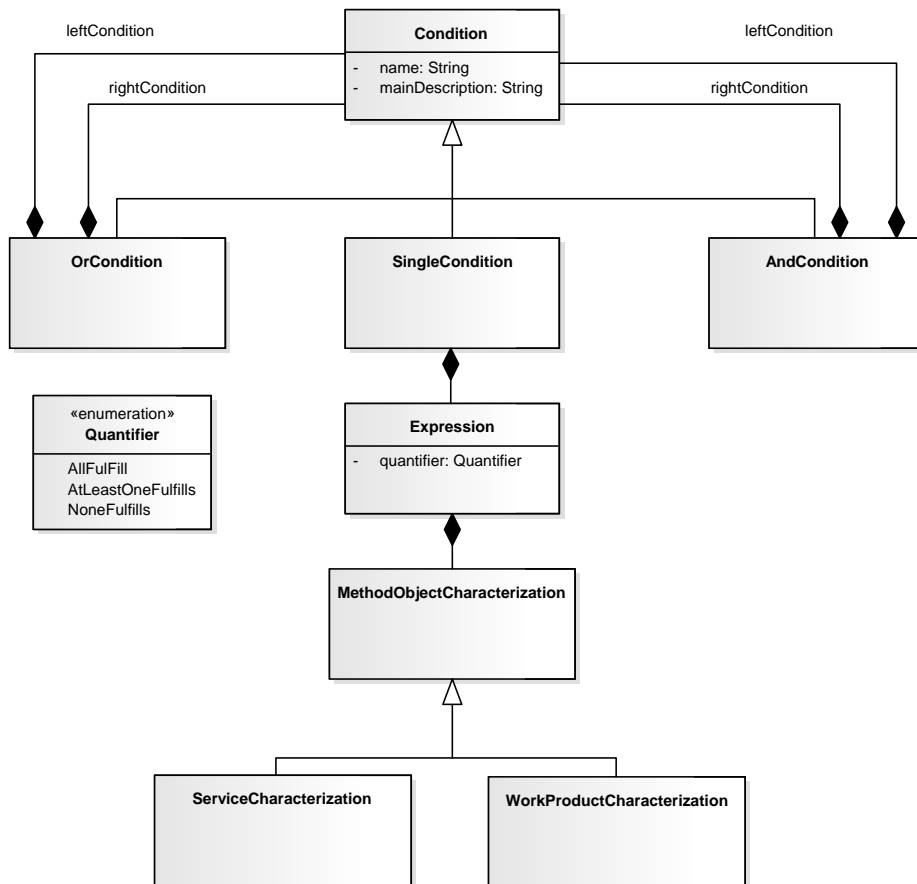


Fig. 4.11 The condition related meta-classes of our meta-model package patternElements

OrCondition As illustrated in Figure 4.11, **OrCondition** is one of the three kinds of conditions. It consists of two conditions, where at least one must be fulfilled in order for the **OrCondition** to be fulfilled.

AndCondition **AndCondition** is the second of the three kinds of conditions. It consists of two conditions that both must be fulfilled in order for the **AndCondition** to be fulfilled.

SingleCondition SingleCondition is the third of the three kinds of conditions. It consists of an expression that is either fulfilled or not fulfilled by the ConstrainedScope that hosts the SingleCondition.

Expression Expression represents an expression about one or more method services or work products that is fulfilled or not. It contains a MethodObjectCharacterization that characterizes method services or method patterns. In addition, it has a *quantifier* that denotes that the MethodObjectCharacterization has to be true for all (AllFulfill), at least one (AtLeastOneFulfills), or no (NoneFulfills) method service or work product in order for the expression to be fulfilled.

MethodObjectCharacterization MethodObjectCharacterization represents a characterization of one or more method services (ServiceCharacterization) or work products (WorkProductCharacterization) that is fulfilled or not.

MethodServiceCharacterization MethodServiceCharacterization represents a characterization of one or more method services. The related meta-classes are illustrated in Figure 4.12.

WorkProductCharacterization WorkProductCharacterization represents a characterization of one or more work products. The related meta-classes are illustrated in Figure 4.13.

OrServiceCharacterization As illustrated in Figure 4.12, OrServiceCharacterization is one of the three kinds of service characterizations. It consists of two service characterizations, where at least one must be fulfilled in order for the OrServiceCharacterization to be fulfilled.

AndServiceCharacterization AndServiceCharacterization is the second of the three kinds of service characterizations. It consists of two service characterizations that both must be fulfilled in order for the AndServiceCharacterization to be fulfilled.

SingleServiceCharacterization SingleServiceCharacterization is the third of the three kinds of service characterizations. It consists of a ServiceIdentificationConstraint that characterizes method services.

ServiceIdentificationConstraint ServiceIdentificationConstraint characterizes method services by identity, category, or usage of work products.

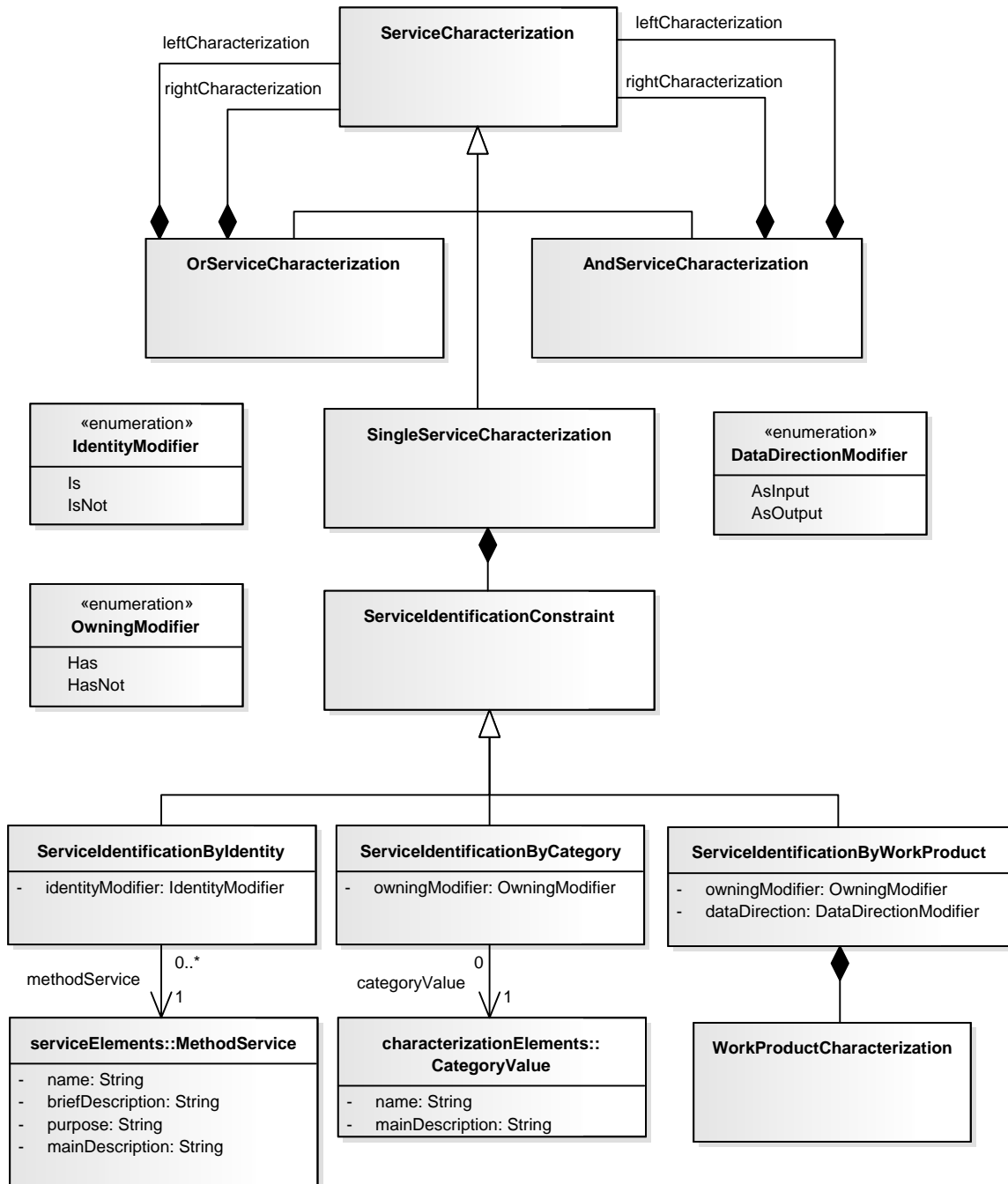


Fig. 4.12 The service characterization related meta-classes of our meta-model package `patternElements`

ServiceIdentificationByIdentity `ServiceIdentificationByIdentity` characterizes method services by identity. It references a method service and requires that one or more method service descriptors for that method service are present or not present, depending on the *identityModifier* being set to `Is` or `IsNot`, respectively.

ServiceIdentificationByCategory `ServiceIdentificationByCategory` characterizes method services by their associated category. It requires that one or more method service descriptors that reference method services with that category are present or not present, depending on the *identityModifier* being set to `Is` or `IsNot`, respectively.

ServiceIdentificationByWorkProduct `ServiceIdentificationByWorkProduct` characterizes method services by their usage of work products. It requires that one or more method service descriptors use a specific work product are present or not present, depending on the *identityModifier* being set to `Is` or `IsNot`, respectively. With the attribute *owningModifier* it is set whether the work product has to be used as input (`AsInput`) or output (`AsOutput`). Instead of specifying a work product directly, the work product is described with the contained `WorkProductCharacterization` as this is more flexible.

OrWorkProductCharacterization As illustrated in Figure 4.13, `OrWorkProductCharacterization` is one of the three kinds of work product characterizations. It consists of two work product characterizations, where at least one must be fulfilled in order for the `OrWorkProductCharacterization` to be fulfilled.

AndWorkProductCharacterization `AndWorkProductCharacterization` is the second of the three kinds of work product characterizations. It consists of two work product characterizations that both must be fulfilled in order for the `AndWorkProductCharacterization` to be fulfilled.

SingleWorkProductCharacterization `SingleWorkProductCharacterization` is the third of the three kinds of service characterizations. It consists of a `WorkProductIdentificationConstraint` that characterizes method services.

WorkProductIdentificationConstraint `WorkProductIdentificationConstraint` characterizes work products by identity and their usage by method services.

WorkProductIdentificationByIdentity `WorkProductIdentificationByIdentity` characterizes work products by identity. It references a work product and requires

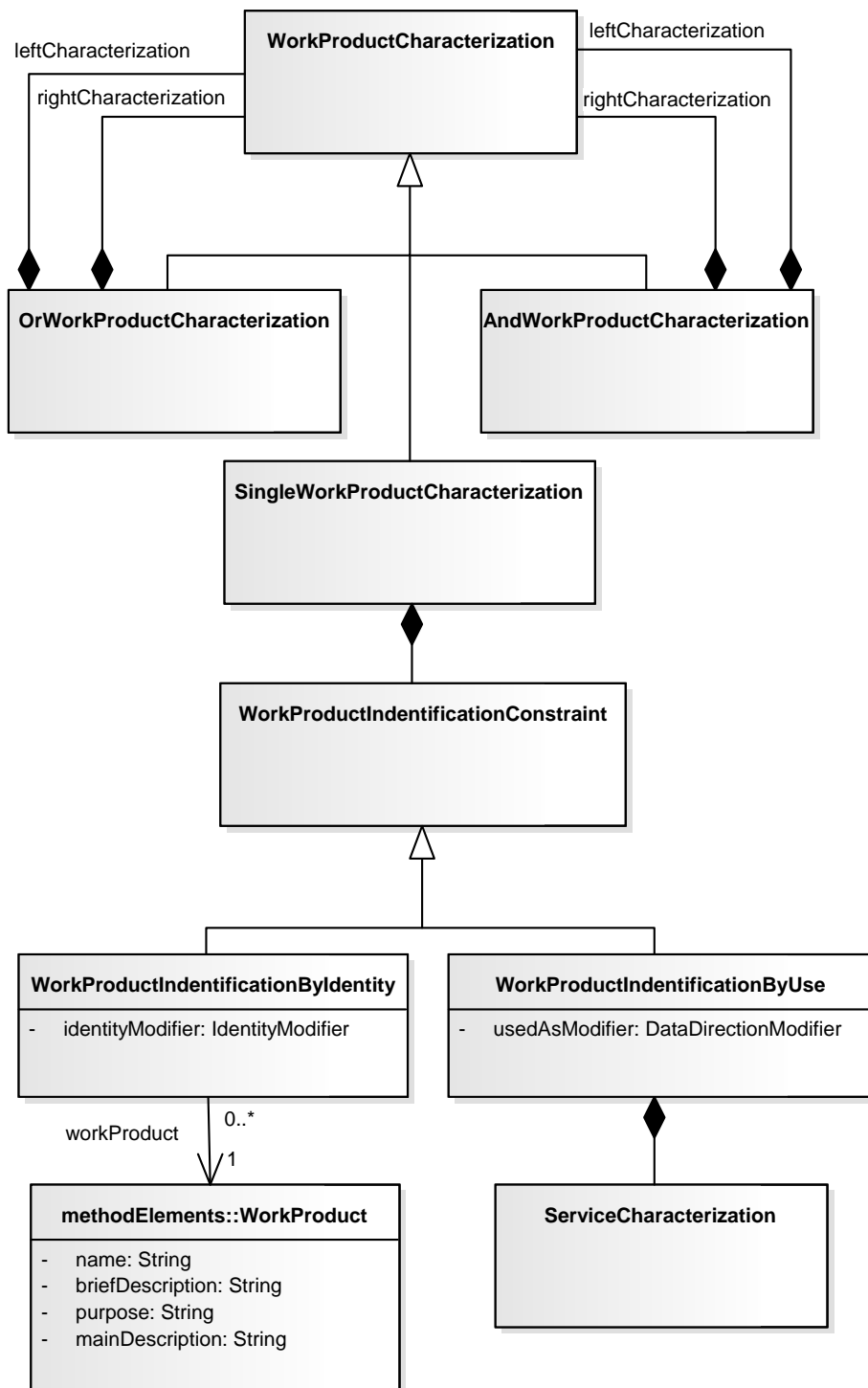


Fig. 4.13 The work product characterization related meta-classes of our meta-model package `patternElements`

that it is used or not, depending on the *identityModifier* being set to Is or IsNot, respectively.

WorkProductIdentificationByUse `WorkProductIdentificationByUse` characterizes work products by their usage within method services. Depending on the *usedAsModifier* attribute, it requires that work products are used as input (`AsInput`) or output (`AsOutput`) of specific method services. Instead of specifying a method service directly, the method service is described with a `ServiceCharacterization` as it is more flexible.

Usage

With our formalization of method patterns, we enable the modeling of abstract orderings with varying levels of complexity and with a high-level modeling language. Figure 4.14 shows the object model for the method pattern derived from Scrum that is illustrated in Figure 3.12. This method pattern is more complex and contains three constrained scopes within a sequence. Our formalization allows defining method patterns that are independent of concrete method services and that can be precisely evaluated for their fulfillment during the quality assurance of composed method models. The constraints in the figure characterize method services mostly by category values. Only the method service *Hold Standup Meeting* is referenced directly. This example also shows that method patterns reuse basic elements to express the usage of work products, situational factor values, and category values.

4.5 Summary

In this chapter, we presented the details of the method content definition with our approach that fulfills the MDRs explained in Section 4.1. In particular, we contributed two ways to extract reusable method content (MDR₁), first, by extraction from methods described in literature, and second, by extraction from the daily practice of organizations. We then explained the formalization of basic elements, of basic method elements (MDR₂) and basic characterization elements (MDR₃). Thereafter, we showed how method services are formalized, our main type of method building blocks that is a reusable, compositional, interoperable, and executable unit of method based on the service-oriented paradigm (MDR₅,MDR₆). Furthermore, we explained the formalization of method patterns, our second type of method building block that captures abstract orderings of activities as a guidance for the project method engineer (MDR₄). We showed, that both, the method service and the method pattern is formalized based on reusing existing meta-classes in order to support consistency (MDR₇).

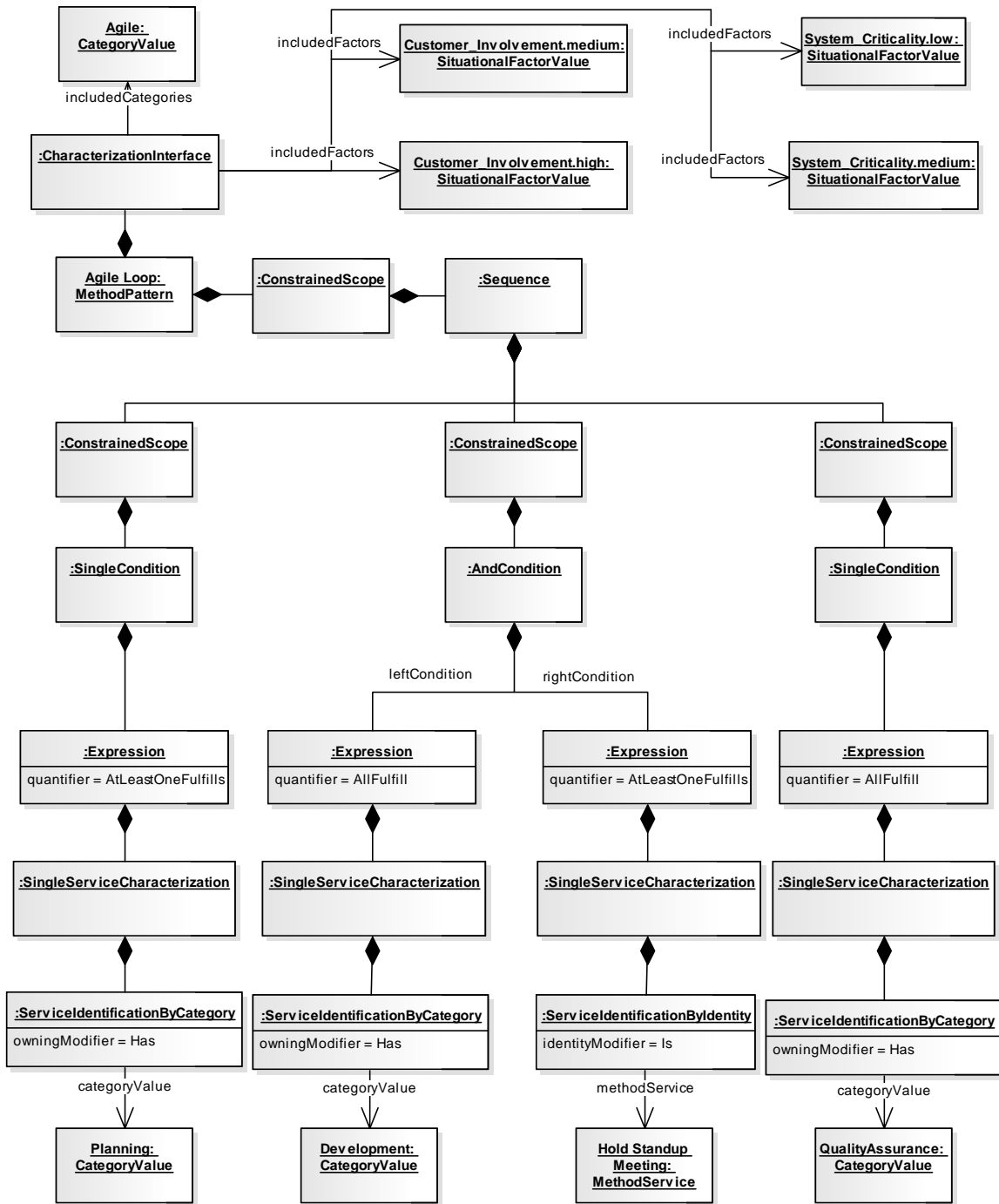


Fig. 4.14 The object model of the method pattern derived from Scrum shown in Figure 3.12

In the following chapter, we discuss the details of method tailoring with our approach and explain requirements and related work, the formalization with the meta-model, and the operationalization with algorithms.

CHAPTER 5

Method Tailoring

In the previous chapter, we presented the method content definition by the senior method engineer. In this chapter, we discuss the details of method tailoring by the project method engineer.

This chapter is structured as follows. In Section 5.1, we discuss the requirements and related work. In Section 5.2, we explain the formalization of project situations. We focus on the definition of method models in Section 5.3. In Section 5.4, we discuss our quality model and quality assurance framework. Then we present the preparation of method models for execution in Section 5.5. Finally, we summarize the chapter in Section 5.6.

5.1 Requirements and Related Work

5.1.1 Requirements

5.1.2 Related Work

5.2 Characterize Project

5.2.1 Meta-Classes

5.2.2 Usage

5.3 Compose Project-Specific Method

5.3.1 Identifying Suitable Method Building Blocks

5.3.2 Specification of Methods

5.4 Assure Quality of Method

5.4.1 Quality Model

5.4.2 Automated Quality Assurance Framework

5.4.3 Usage

5.5 Initialize Method

5.5.1 Transformation, Deployment & Configuration

5.5.2 Usage

5.6 Summary

5.1 Requirements and Related Work

In this section, we describe the requirements and related work of method tailoring. We first present the method tailoring requirements (MTRs) that are a refinement of the solution requirements (SRs) presented in Section 2.2.1. Then we briefly summarize the related work that will be discussed also in the respective sections later.

5.1.1 Requirements

In this section, we discuss the requirements with respect to the method tailoring for a holistic solution for software engineering method management based on an assembly-based method engineering approach.

In assembly-based method engineering, suitable method building blocks need to be composed based on the project situation. According to SR2.1, the first task of a project method engineer is therefore to characterize the project situation, so that she can determine suitable method building blocks based on it. The first MTR for our solution is therefore to *enable the formalization of project characteristics such that suitable method building blocks can be determined (MTR₁)*.

After the project is characterized, the project method engineer has to compose the project-specific method model by composing selected method building blocks. In order to provide her with guidance (SR2.2), a MTR is to *enable her to combine selected method patterns by nesting them into each other (MTR₂)*.

Also important for the composition of method models is to enable their executability (SR3.1). Therefore, a MTR is that our formalization needs to *support formal control and data flow in method models (MTR₃)*. Based on SR3.1, it should allow the project method engineer to use a high-level language to formalize both.

A project method engineer has to be able to check for the consistency of the composed method model (SR2.4). As the notion of quality is generally not defined for software engineering method models, a MTR is to *define and categorize the notion of quality for software engineering method models (MTR₄)*.

As method models can become so large that the manual quality assurance is very tedious, another MTR is to *automatically check for the consistency of method models*

(*MTR*₅). The analysis has to run fast enough to be used during method composition by the project method engineer. It has to check for the executability of method models and the fulfillment of used method patterns and shall be extensible.

Composed method models shall be executable (SR3.1). Instead of implementing our own process engine, we decide to enable the use of process engines established as standard in industrial practice. Therefore, each of our method models has to be transformed into a BPEL/BPEL4People process model for its execution. Thus, a MTR is to *transform method models to process models automatically* (*MTR*₆).

The transformed process model has to provide an interface for the team members to interact with the method (SR3.2). Therefore, a MTR for the transformation from method models to process models is to *incorporate the creation of suitable GUIs for tasks into the transformation* (*MTR*₇). The team members shall be enabled to access and specify information about work products, tasks, and the method execution state, e.g., the current iteration or project phase.

The discussed MTRs are summarized in Table 5.1. Also illustrated is the MESP task where the specific requirement needs to be addressed. In the following, we discuss each MESP task of our solution for method tailoring. We then also explain how the respective requirements are met.

Table 5.1 Method tailoring requirements and the affected MESP tasks

<i>Requirement</i>	Description	MESP Task
<i>MTR</i> ₁	enable the formalization of project characteristics such that suitable method building blocks can be determined	Characterize Project
<i>MTR</i> ₂	enable her to combine selected method patterns by nesting them into each other	Compose Project-Specific Method
<i>MTR</i> ₃	support formal control and data flow in method models	Compose Project-Specific Method
<i>MTR</i> ₄	define and categorize the notion of quality for software engineering method models	Assure Quality of Method
<i>MTR</i> ₅	automatically check for the consistency of method models	Assure Quality of Method
<i>MTR</i> ₆	transform method models to process models automatically	Initialize Method
<i>MTR</i> ₇	incorporate the creation of suitable GUIs for tasks into the transformation	Initialize Method

5.1.2 Related Work

Regarding the project characteristics, many approaches discuss the use of situational factors and associated values, e.g., [HV97], [NH03] and [KDS07], similar to the characterization of method building blocks. In [GGH08], the authors discuss a slightly different idea by considering so-called *method goals*. However, in these approaches have in common that the situation is determined by selecting one of the provided values for each attribute in a list.

Regarding the identification of suitable method building blocks, most tool-supported approaches only offer facilities to browse the method repository manually for suitable method building blocks, e.g., [KLR96],[Cer+11],[Ell+11] and only few approaches offer additional support, e.g., by a textual query language [BSH01]. Proposals that go beyond that involve the use of similarity measures and backward chaining of activities based on the required output work products [GH08].

Regarding the formalization of method models only few assembly-based method engineering approaches allow modeling methods that are executable, e.g., Demacrone [Har97] and MENTOR [SRG96],[Pli96]. More recently proposed standard meta-models like SEMDM and SPEM lack support for executability. Extensions to these standards that enhance them for executability exist, e.g., [Ben+07] and eSPEM [Ell+10; Ell+11], however, they lack support for assembly-based method engineering.

Regarding the analysis of method models for quality issues, there is no accepted and standardized notion of quality for software engineering method models. Only few works discuss measurable and quantifiable notions of quality and propose well-formedness rules or quality constraints, e.g., [Har97; BSH98], [Chroo], and [Per+11].

Regarding the initialization of method models for enactment, some approaches focus on creating documentation for the method to be used by the project team, e.g., EPF Composer, IBM Rational Method Composer, and MC Sandbox [KÅ11; KÅ12], while others derive a suitable Computer-aided software engineering (CASE) environment [PS97], [KLR96], [Cer+11]. Only few approaches discuss execution support, either supported natively by build-in execution support [Har97], [SRG96; Pli96], [Ell+10; Ell+11], or by a mapping to an executable process description language with process engine support [Ben+07].

5.2 Characterize Project

In this section, we discuss how our solution addresses the formalization of project characteristics to support the search for method building blocks (MTR₁). In the following, we first discuss related work and then explain our formalization.

The role of project characteristics and their formalization is discussed for example in [HV97]. Here the author proposes to determine the situation by assessing situational factors of the project. These should then be used to query for suitable method building blocks. However, a solution is not proposed. In [NH03], the authors discuss tool support for the OPEN approach [FHo2] and propose the use of a set of situational factors and associated values. They list a first set of 16 situational factors, e.g., *domain* and *product size* and propose that these can best be elicited using a questionnaire. In [KDS07], the authors extend the assembly-based approach first proposed in [RR01] with the assessment of project characteristics. They also propose the use of situational factors and associated values. They discuss the four dimensions *organizational*, *human*, *application domain*, and *development strategy*, where each dimension contains several situational factors and their possible values. Related work specifically on situational factors, i.e., [Bek+08] and [CO12], is discussed in Section 4.3. In [GGHo8], the authors discuss a slightly different idea by considering so-called *method goals*, e.g., *product reliability* or *project time constraints*, and then assign each method building block values that expresses its contribution, e.g., *enhances* or *deteriorates*. They propose the use of goal analysis (e.g. [Bre+04]) to find the best combination of method building blocks with respect to the prioritized method goals. However, they only explain how to determine the better of two sets of method building blocks, not how to come up with the set in the first place.

In the following, we explain the formalization of the project characteristics with our software engineering method meta-model and then illustrate its usage.

5.2.1 Meta-Classes

In this section, we discuss the meta-classes of our MESP meta-model used to formalize the project characteristics as illustrated in Figure 5.1.

ProjectMethod `ProjectMethod` represents the method model throughout its composition and contains all other method model related elements. Thus, the formalization of the project characteristics is modeled to be part of `ProjectMethod`: it contains a `ProjectGoal` and a `ProjectSituation`. In addition, `ProjectMethod` contains attributes that allow giving it a name and descriptions.

ProjectGoal `ProjectGoal` is one of the two meta-classes that is used in our approach to formalize the project characteristics. Our notion of project goal differs from the goals used in [GGHo8] as there the focus was on prioritizing among different qualities when choosing a set of method building blocks. However, our project goal captures what needs to be produced by the method in the project and what input is available at project start to do that. Thus, it is associated with input and output `Work Products`. It also associates the `Roles` that perform work

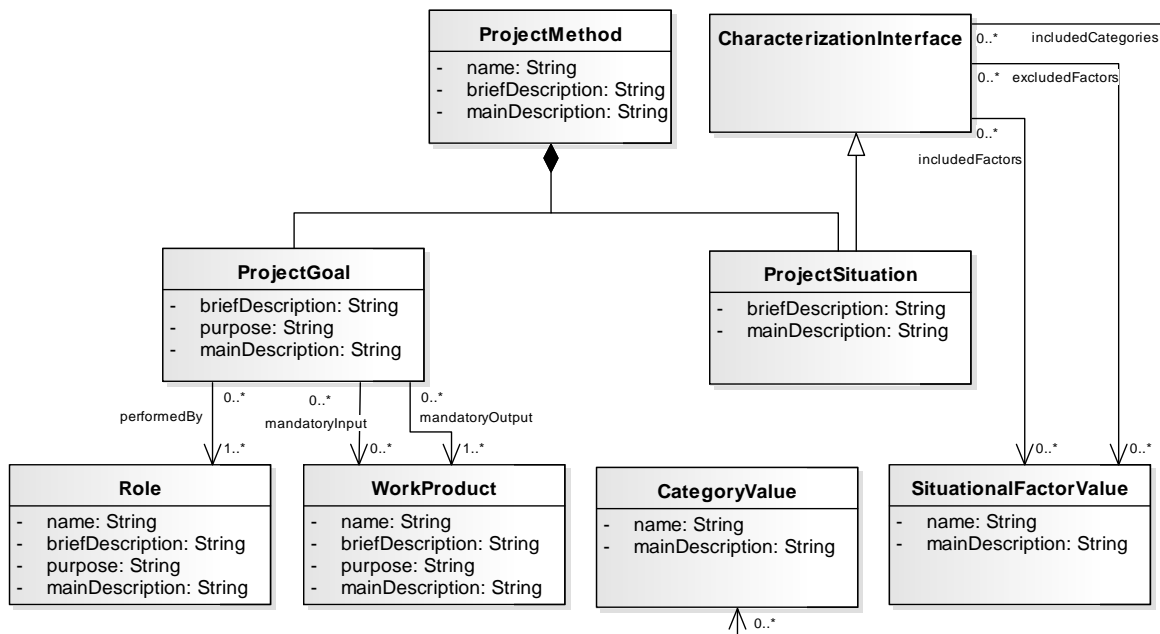


Fig. 5.1 The project characteristics related meta-classes of our meta-model package ProjectMethod

in the method model. While roles might not always be relevant as a requirement for project composition, the list of involved roles is helpful after the method is composed, for example during method initialization. `ProjectGoal` has attributes that allow giving it a descriptions and to explain the purpose of the method model.

ProjectCharacterization `ProjectCharacterization` is the second of the two meta-classes that is used in our approach to formalize the project characteristics. It expresses the situation by sets of situational factor values and characterization values following the same idea as other approaches discussed in related work. The associations are established by inheritance from the meta-class `CharacterizationInterface`. In addition, `ProjectCharacterization` has attributes that allow giving a textual description of the situation.

5.2.2 Usage

Our formalization of the project characteristics allows us to capture textual descriptions of the project goal and the project characterization. More importantly, it allows us to formally associate work products, roles, situational factor values, and characterization values as we illustrated with Figure 3.18. The formal description eases the search for suitable method building blocks, but also benefits the quality assurance and initialization of the composed method model. While many approaches propose a fixed list of possible work products or situational factor values, our

solution allows the senior method engineer to add new basic elements and the project method engineers to directly use them during project characterization.

5.3 Compose Project-Specific Method

In this section, we discuss how suitable method building blocks are identified with our solution and how they are composed to executable method models (MTR₃) including support for combined patterns (MTR₂).

Regarding the identification of suitable method building blocks, many tool-supported approaches only allow manually browsing the method repository, e.g., [KLR96], [Cer+11],[Ell+11]. The tool Demacrone [Har97] additionally offers the textual query language MEL [BSHo1] that is similar to SQL and allows querying for method building blocks that fulfill certain criteria, e.g., `SELECT X Which PRODUCES Object Model and PARTICIPATION_TYPE = Much user participation`. The tool MENTOR [SRG96; Pli96] also supports the identification of suitable method building blocks by presenting a list of candidate building blocks based on the situation and the context of a task. In [RRo1], four generic strategies for the selection of method building blocks are discussed. The evaluation strategy describes the approach, where the degree of matching between the candidate method building block to the project characteristics is used. This can be calculated using similarity measures as also described in [MRo6]. The decomposition strategy can be used, if a method building block is a composite building block. It proposes to deselect building block parts that are not required to fulfill the current requirements, i.e., the project characteristics. The aggregation strategy proposes to look for composite building blocks that contain the candidate building block, if it does not fulfill the requirements fully. The refinement strategy proposes to look for a different building block that also fulfills the requirements, but offers richer content, e.g., by having more guidances. In [GHo8], the authors propose to create a method model by backward chaining the activities that produce the required output. Thus, based on the final work product of the method, the activities that produce these work products are identified and added as predecessors. Then for the required input work products of these predecessors, suitable activities are searched and added. This is done recursively until activities are added, where the required input work products are available at project start.

Regarding the formalization of method models, only few assembly-based method engineering approaches allow modeling methods that are executable. Demacrone [Har97] and MENTOR [SRG96],[Pli96] support the creation of executable method models that are composed of method building blocks. However, both are based on outdated, proprietary technology, low-level specification languages and low-level (e.g. character-based) user interfaces. More recently proposed standard meta-models for software engineering methods like SEMDM and SPEM

lack execution tool support (SEMDM) or execution support in general (SPEM). Bendraou et al. [Ben+07] and Ellner et al. [Ell+10; Ell+11] propose extensions to SPEM to address SPEMs lack of executability. Bendraou et al. propose xSPEM and present ideas to map it to BPEL in order to provide enactment support and sketch some mappings between concepts of xSPEM and BPEL [Ben+07]. They discuss that BPEL₄People could be used to allow for human interaction, however no details are given. Also, assembly-based method engineering is not addressed. Ellner et al. developed eSPEM [Ell+10] and provide tooling based on the Eclipse ecosystem to model and execute software engineering methods [Ell+11]. However, situational method engineering support is very limited as method building blocks and the project cannot be characterized. Other approaches focus explicitly and solely on the provision of executable method models and corresponding process engines, however, neglecting situational method engineering. Based on the idea of process programs [Ost87], in [Wis+00] software engineering methods are described with Little-JIL, a language for programming the coordination of agents. As this language and the provided tool support focus on the coordination of tasks, the actual enactment of tasks is left to the agents and descriptions and guidances for tasks are not supported.

In the following, we discuss how the project method engineer identifies suitable method building blocks for her project-specific method. Thereafter, we explain the formalization of method model compositions with our software engineering method meta-model and then illustrate its usage.

5.3.1 Identifying Suitable Method Building Blocks

Our approach to identify suitable method building blocks consists of three steps that are performed in parallel and represent three strategies to identify suitable method building blocks. While two are known from related work, one is based on our novel notion of method patterns. In the following, we will discuss each step.

Identify based on Characterization

In our approach, the project situation is characterized with the same situational factor values and category values that are used to characterize method services and method patterns. Therefore, suitable method building blocks can be identified using an evaluation strategy [RR01], i.e., based on the similarity of situational factor values and category values between the project situation and the characterization interface of building blocks. For example, if the project situation includes the situational factor *customer_involvement.medium* then method building blocks with *customer_involvement.medium* among their involved factors can be identified.

Part of this step is also to refine the project situation based on the amount of matching method building blocks. To reduce the number of matching method

building blocks, additional situational factors could be considered. For example, additionally taking into account the system criticality into the project situation with adding *system_criticality.medium* could reduce the number of matching method building blocks. On the contrary, if the search for method building blocks does not render sufficient results, the project characterization can be relaxed by reducing the number of situational factors (e.g. by removing *system_criticality.medium*) or by adding situational factor values that relax this condition, e.g., by adding *criticality.low*.

Identify based on Work Products

In our approach, the project goal is characterized with input and output work products and the roles to be involved via the method model. In addition, each method service describes the input and output work products and responsible role as part of its structural interface. Following the idea proposed in [GHo8], suitable method building blocks can thus be identified by their provided output work products based on the currently needed work products. For example, when integration test results are required as an output of the method, the project method engineer can check the method repository for method services that provide that output. This strategy is not limited to the required output work products of the method, but can be used also to add method services that create the output work products needed as input by method services used in the method model.

We propose additionally using the provided input work products that are available at project start to identify method services that process them towards the output work products. For example, if a requirements specification is provided at project start or created by a method service in the method model, the method service *Create Test Cases* could be added, as it uses a requirement specification as input. While it might turn out that this identified method service is not useful to progress towards the required outputs of the method model, this strategy can help to identify useful method services. It is not limited to the provided inputs at project start, but can be used also to add method services that consume an work product that is created by method services used in the method model.

Identify based on Method Pattern Constraints

As our solution for software engineering method management introduces the notion of method patterns, we propose a third strategy to identify suitable method building blocks. Method patterns used in the method model are fulfilled, when the condition of all of their constraint scopes is fulfilled. These conditions describe the necessity for the presence or absence of method services based on their identity (specifically named), associated category values, or processed or produced work products. Therefore, useful method services can be identified by searching for

method services that fulfill the conditions. For example, the first constrained scope shown in Figure 3.15 is fulfilled with a method service that has a system architecture as its output work product. Consequently, the method service *Prepare System Specification* can be identified and a corresponding method service descriptor can be added to the method model.

5.3.2 Specification of Methods

As SPEM is the most widespread software engineering method meta-model with available tool-support, we used it as the foundation for our basic method elements. However, as SPEM lacks execution-support and support for our method services and method patterns, we cannot use it or extensions like eSPEM [Ell+10] to model composed method models that are executable. Therefore, we introduce our own composition language to describe composed, executable method models, however, we rely on the two standards SPEM and BPEL. SPEM is the de facto standard for software engineering methods, however without support for executability. BPEL is one of the two de facto standard executable process description languages with broad and mature tool support, however without support for the typical terminology of software engineering methods and with a textual and very technical specification language. Our terminology is influenced by the Process-related classes of SPEM (cf. Figure 4.4), while we adopted the structure of BPEL processes. Similar to the idea of xSPEM [Ben+07], we provide a mapping of our language to BPEL to give it execution semantics. However, our approach goes beyond xSPEM, as we also support data flow and GUIs for the project team members. Hence, we also rely on BPEL4People. Our formalization of method models allows specifying executable compositions of method building blocks with deterministic control and data flow. The formalization is based on the scalable well-formed structure of BPEL process models, but allows the project method engineer to use a high-level language with the typical software engineering method related terminology.

Meta-Classes

In this section, we discuss the method-composition related meta-classes of our MESP meta-model that are adopted from SPEM and BPEL. Figure 5.2 illustrates the control-flow related meta-classes of our meta-model, while Figure 5.3 illustrates the data-flow related meta-classes. Some associations are omitted for better readability.

Activity An **Activity** represents a general unit of work within a **Process**. It is sub-typed either to describe elemental steps of the process behavior or to describe deterministic control-flow logic. Therefore, **Activity** follows BPEL's notion rather than the notion of activity in SPEM. In SPEM, activities group other elements only in terms of breakdown structures. As illustrated, **Activity** contains attributes that

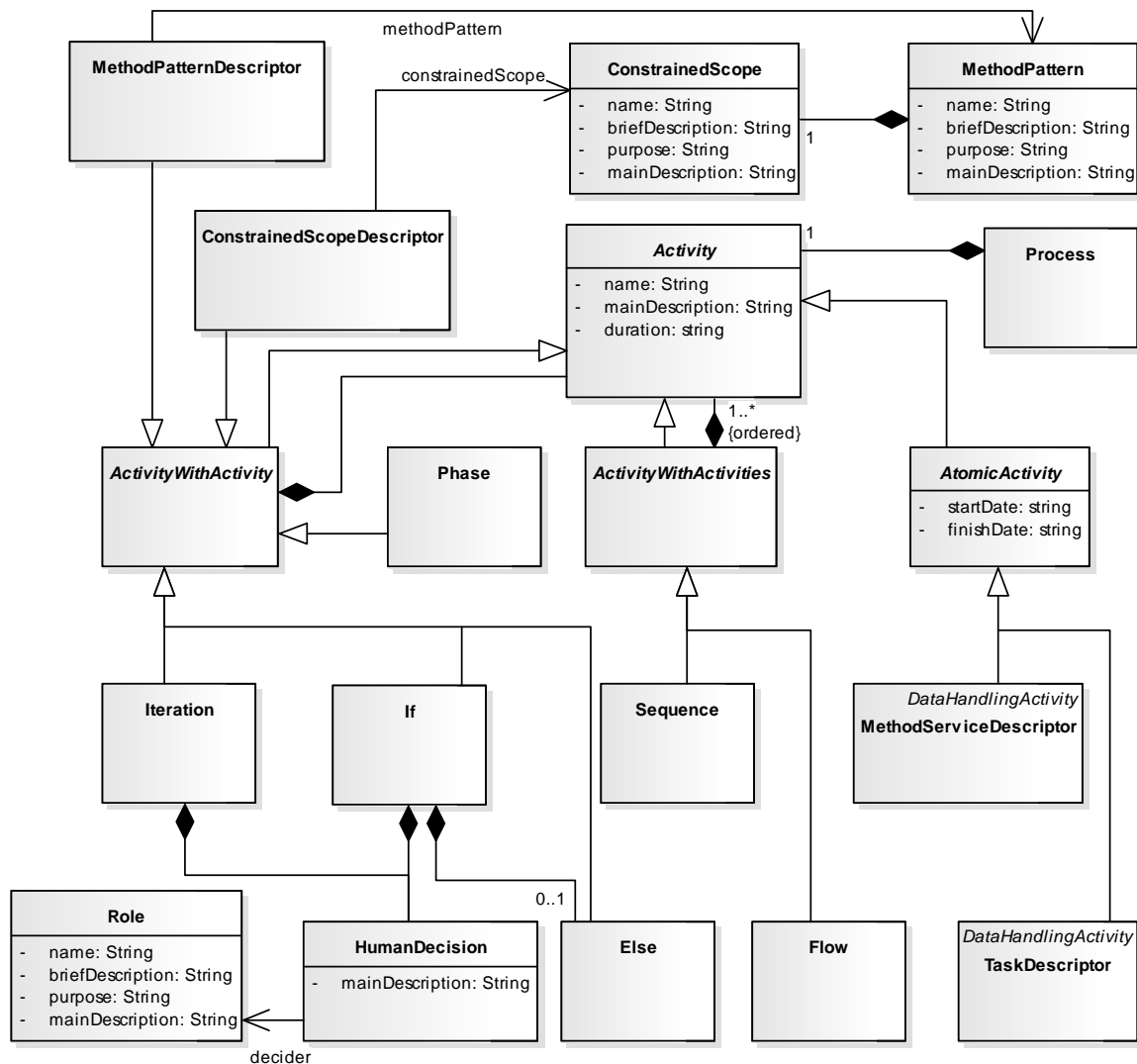


Fig. 5.2 The control-flow related method elements of our meta-model package Project-Method

allow giving it a name and a textual description. In addition, similar to planning data in SPEM, a duration can be given as an exact recommended duration or a ratio.

AtomicActivity AtomicActivity inherits from Activity and represents an elemental step of the process behavior as it is sub-typed by MethodServiceDescriptor and TaskDescriptor. As illustrated, AtomicActivity contains attributes that allow giving it a planned start and finish date similar to planning data in SPEM.

MethodServiceDescriptor MethodServiceDescriptor represents the use of a method service within a specific point in the control flow of a process, either the

process of a method model or the process of a method service that represents a composite activity. The use of `MethodServiceDescriptor` as a proxy element within a process resembles the usage of `Task` and `Task Use` within SPEM.

TaskDescriptor `TaskDescriptor` represents the use of a task within a specific point in the control flow of a process, either the process of a method model or the process of a method service. Typically, task descriptors are meant and should be used for the latter, as tasks do not have an interface description as opposed to method services. The use of `TaskDescriptor` as a proxy element within a process resembles the usage of `Task` and `Task Use` within SPEM.

ActivityWithActivities `ActivityWithActivities` inherits from `Activity` and represents deterministic control-flow logic that specifies how the contained activities are executed. For this purpose, it is sub-typed by `Sequence` and `Flow`.

Sequence `Sequence` contains one or more activities that are performed sequentially, in the lexical order of their containment within the `Sequence` element, following the semantics of BPEL. The `Sequence` completes when the last activity in the sequence has completed. If the `HumanDecision` of an activity evaluates to false then it is skipped and also considered completed.

Flow `Flow` contains one or more activities that are performed in parallel, following the semantics of BPEL. Thus, a flow enables concurrency. The `Flow` completes when all of the activities within the flow have completed. If the `HumanDecision` of an activity evaluates to false then it is skipped and also considered completed. Thus, a flow also enables synchronization.

ActivityWithActivity `ActivityWithActivity` inherits from `Activity` and is sub-typed by activities that can contain only one activity. These describe either deterministic control-flow logic that specifies how the contained activity is executed (i.e. `Iteration`, `If`, and `Else`), add semantic structure to the process (i.e. `Phase`), or describe the usage of method patterns. If multiple activities should be placed within an `ActivityWithActivity` then an `ActivityWithActivities` has to be used to specify their control-flow order.

Iteration `Iteration` inherits from `ActivityWithActivity`. Adopting the definition in SPEM, it represents an important structuring element to organize work in repetitive cycles. In addition, `Iteration` follows the execution semantics of `RepeatUntil` in BPEL and provides for repeated execution of a contained activity. The contained activity is executed until its contained `HumanDecision` evaluates to

false. The `HumanDecision` is tested after each execution of the body of the iteration. The iteration executes the contained activity at least once.

If, Else `If` inherits from `ActivityWithActivity` and follows the execution semantics of BPEL. It provides conditional behavior. The activity consists of one conditional branch defined by the `If` element, followed by an optional `Else` element. If the `HumanDecision` of the `If` evaluates to true, the branch is taken, and its contained activity is performed. Otherwise, the `Else` branch is taken, if present. The `If` activity is completed, when the contained activity of the selected branch completes, or immediately, when the `HumanDecision` evaluates to false and no `Else` branch is specified.

HumanDecision `HumanDecision` represents a Boolean decision that has to be taken by a project team member and that decides whether a certain activity is executed or not. The group of people that is requested to take the decision is determined by the referenced role. `HumanDecision` has an attribute that allows providing textual background for the decision.

Phase `Phase` inherits from `ActivityWithActivity`. Adopting the definition in SPEM, it represents a significant period in a project, typically ending with activities that represent a major management checkpoint or produce a set of major work products. A `Phase` completes when its contained activity has completed.

MethodPatternDescriptor `MethodPatternDescriptor` represents the use of a method pattern within a specific point in the control flow of a process. It has an association to the method pattern that it represents. While method services can be represented with atomic activities, a `MethodPatternDescriptor` has to allow the method engineer to specify what method service descriptors shall be used within the contained constrained scopes. Thus, similar to `MethodPattern`, a `MethodPatternDescriptor` is an `ActivityWithActivity`. It should contain the same structure as the content of the corresponding method pattern; however, for every constrained scope of the method pattern it has to contain a corresponding `ConstrainedScopeDescriptor` that represents a reference to it. Our formalization also allows combining two method service descriptors by nesting the one into the other. The use of `MethodPatternDescriptor` as a proxy element within a process resembles the usage of `Task` and `Task Use` within SPEM.

ConstrainedScopeDescriptor `ConstrainedScopeDescriptor` represents the use of a constrained scope of a method pattern within a specific point in the control flow of a process. It has an association to the constrained scope that it represents and

that specifies the constraint. Like method pattern descriptors, a `ConstrainedScopeDescriptor` has to allow the method engineer to specify what method service descriptors (or activities in general) shall be used within it. Thus, similar to `ConstrainedScope`, a `ConstrainedScopeDescriptor` is an `ActivityWithActivity`. It should contain the same structure as the content of the corresponding constrained scope. The use of `ConstrainedScopeDescriptor` as a proxy element within a process resembles the usage of `Task` and `Task Use` within SPEM.

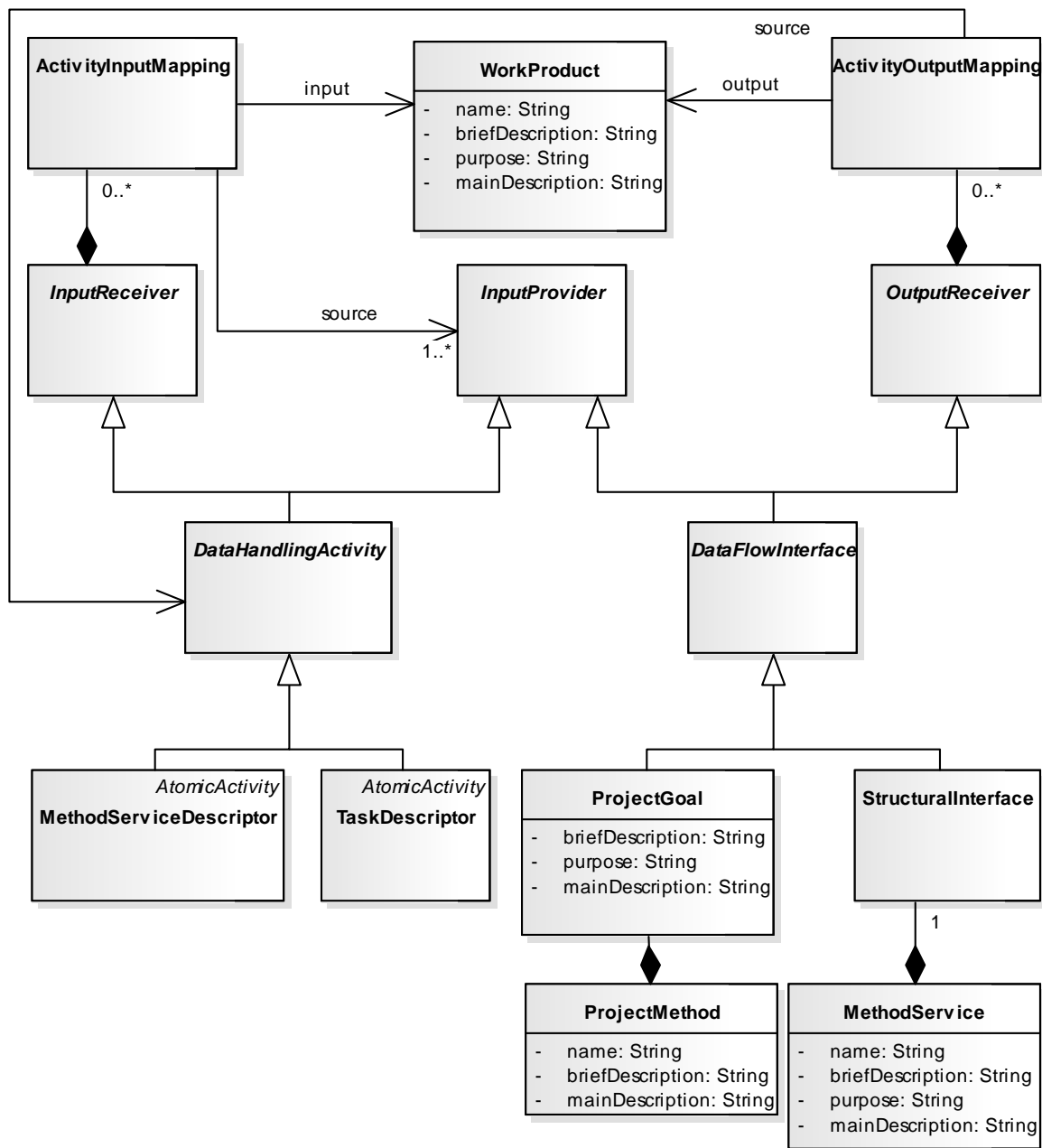


Fig. 5.3 The data-flow related method elements of our meta-model package `ProjectMethod`

ActivityInputMapping The data flow related meta-classes are illustrated in Figure 5.2. While languages like BPEL use the assignment to variables to specify data flow, we directly specify source and targets of data flow. We believe that this is easier to handle for the method engineers, as it is easier to specify and analyze. `ActivityInputMapping` represents the data flow within a process. It specifies from where an activity receives its input work products. It is contained by an `InputReceiver`, which is sub-classed by `MethodServiceDescriptor` and `TaskDescriptor` and specifies the target of the data flow. The association to `WorkProduct` specifies what type of work product is received and the association to the `InputProvider` specifies the sources of the data flow. `InputProvider` is not only sub-classed by `MethodServiceDescriptor` and `TaskDescriptor`, but also by the interfaces of method models and method services. Thus, it can be specified that the input of an activity is provided via the interface as an input of the process.

InputReceiver `InputReceiver` is an abstract meta-class and represents an activity that can receive input work products. It contains activity input mappings to specify from where these inputs are coming. It is sub-classed by `DataHandlingActivity`, which in turn is sub-classed by `MethodServiceDescriptor` and `TaskDescriptor`.

InputProvider `InputProvider` is an abstract meta-class and represents an activity or interface that can offer work products as inputs for other activities. It is indirectly sub-classed by `MethodServiceDescriptor`, `TaskDescriptor`, `ProjectGoal`, and `StructuralInterface`.

ActivityOutputMapping `ActivityOutputMapping` represents the data flow from within a process to its interface. It specifies from where an interface receives its output work products. It is contained by an `OutputReceiver`, which is sub-classed by `ProjectGoal` and `StructuralInterface` and specifies the target of the data flow. The association to `WorkProduct` specifies what type of work product is received. The association to the `DataHandlingActivity` specifies the sources of the data flow and is sub-classed by `MethodServiceDescriptor` and `TaskDescriptor`.

OutputReceiver `OutputReceiver` is an abstract meta-class and represents an interface that can receive output work products from within the process. It contains activity output mappings to specify from where these outputs are coming. It is sub-classed by `DataFlowInterface`, which in turn is sub-classed by `ProjectGoal` and `StructuralInterface`.

DataHandlingActivity `DataHandlingActivity` is an abstract meta-class and represents activities that can produce and consume work products. It is the source for

activity output mappings and inherits the association to be the source of activity input mappings. It is sub-classed by `MethodServiceDescriptor` and `TaskDescriptor`.

DataFlowInterface `DataHandlingActivity` is an abstract meta-class and represents interfaces that can provide and consume work products. Via inheritance, it owns activity output mappings and is the source of activity input mappings. It is sub-classed by `ProjectGoal` and `StructuralInterface`.

Usage

With our specification, we enable the composition of executable method models that possess deterministic control and data flow. The used method services and method patterns are represented by proxy descriptor elements that allow reusing method building blocks multiple times within the same and within different method models. Additional meta-classes are used to specify the control and data flow.

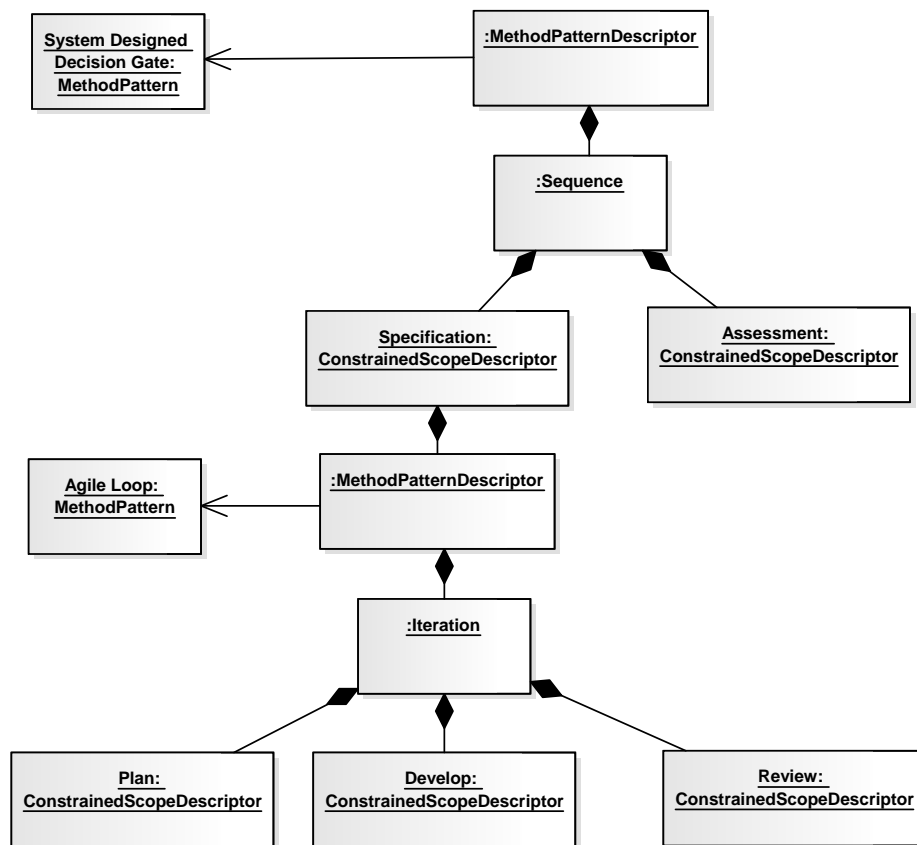


Fig. 5.4 The object diagram for the composed method patterns of Figure 3.19

As part of the end-to-end example in Chapter 3 we showed two nested method patterns in Figure 3.19. Figure 5.4 shows the corresponding object diagram with

its formalization with two nested method pattern descriptors. As illustrated, the method pattern descriptors reference the method patterns from the method repository and contain a constrained scope descriptor for every constrained scope of the method patterns. The constrained scope descriptors are contained by a sequence or iteration respectively. Not depicted in the figure are the references of the constrained scope descriptors to the corresponding constrained scopes and the human decision of the iteration element (see Section 5.3.2). The *Specification* constrained scope descriptor of the method pattern descriptor for the *System Designed Decision Gate* contains the second method pattern descriptor for the *Agile Loop*.

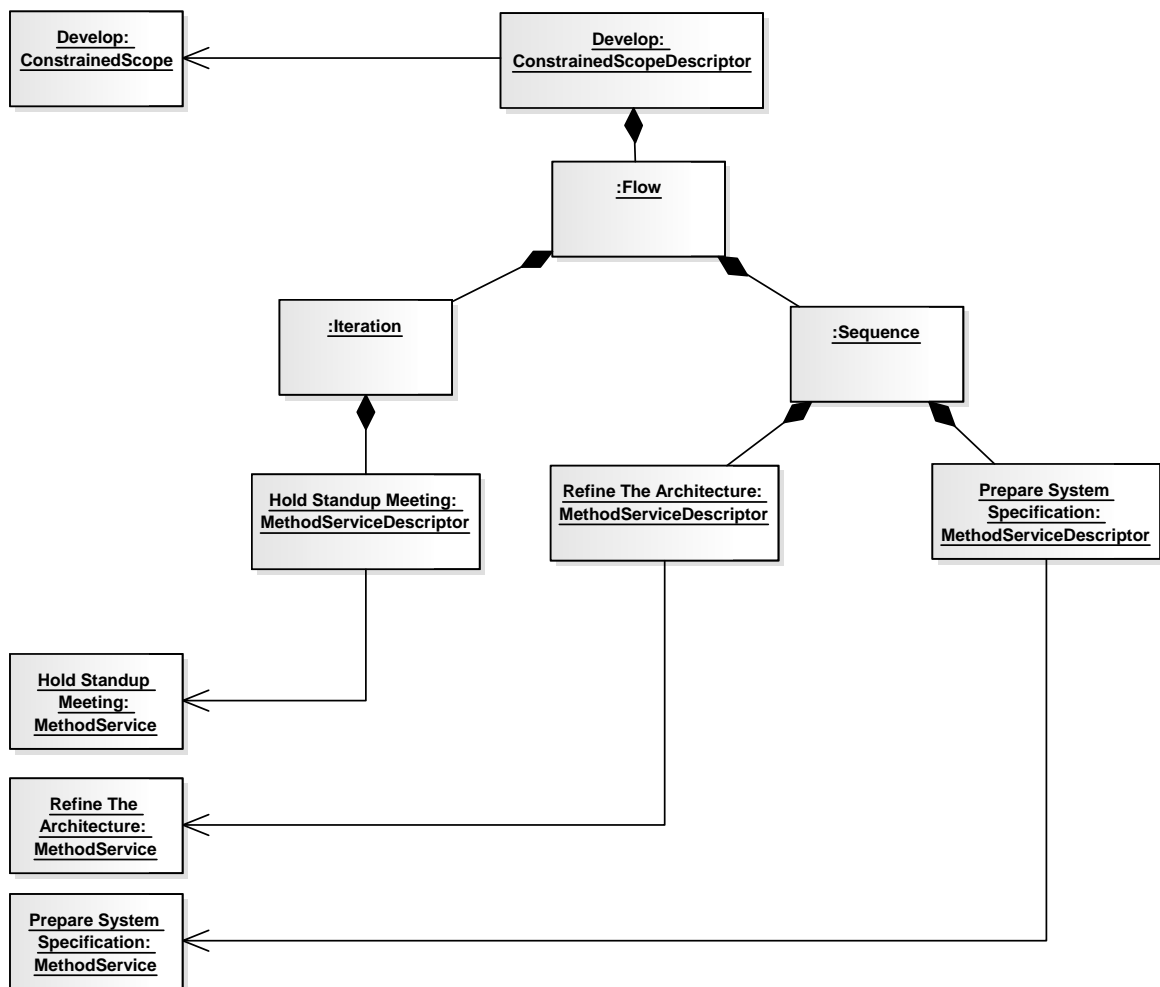


Fig. 5.5 The object diagram for the *Develop* constrained scope of Figure 3.22

The method pattern descriptors and the related elements form a frame to place method service descriptors and to specify refined control flow. Figure 5.5 shows the object diagram for the *Develop* constrained scope of Figure 3.22. As depicted, several method services descriptors were placed into the constrained scope descriptor

and are now contained by it. The flow element specifies that the iteration and the sequence are concurrently executed. Thus, the method service descriptor for *Hold Standup Meeting* is repeatedly executed in parallel to the sequentially executed method service descriptors for *Refine The Architecture* and *Prepare System Specification*. Also shown is that method service descriptors reference the method services that they represent.

The method service descriptor *Refine the Architecture* depicted in Figure 5.5 requires input work products, but does not have an activity input mapping yet (cf. Figure 3.22). Figure 5.6 shows an object diagram with the specification of the data flow with activity input mappings corresponding to Figure 3.22. As shown, the method service descriptor *Refine the Architecture* contains an activity input mapping that points to the source of the data flow, which is the method service descriptor *Envision the Architecture*. It also references the work product *Architecture Notebook* to denote the type of data flow. Please note that several other elements were omitted in the figure for clarity.

5.4 Assure Quality of Method

In this section, we discuss how the project method engineer assures the quality of composed method models. Quality assurance comprises the tasks of defining quality goals, defining how they are measured, and when they are reached. To support this task, we offer an automated quality analysis that reports quality issues and method pattern violations as published in [FK16]. It is in part based on the results of a master thesis [Klu14].

One popular way to assure the quality of methods [Hen+14] is to define a quality model that documents the quality goals and to define metrics on top of it. For example, the predecessor of the ISO/IEC 25010 [ISO11] standard, ISO/IEC 9126-1 [ISO01], defines such a quality model for software product quality. It describes a set of quality characteristics that are refined into sub-characteristics and eventually measurable quality attributes.

As the notion of quality is not generally defined, in this section we present our quality model for software engineering method models (MTR4). We then describe how quality characteristics from this quality model are formalized in order to allow for automated quality assurance checks regarding general quality characteristics and the fulfillment of method pattern descriptors used in the method model (MTR5) and we also discuss our extensible analysis framework as published in [FK16].

Regarding the related work, the quality of methods and method models is discussed in several works; however there is no established quality model for methods nor an established means to measure the quality of method models apart from manually assessing the conformance to capability maturity models like CMMI and Software Process Improvement and Capability Determination

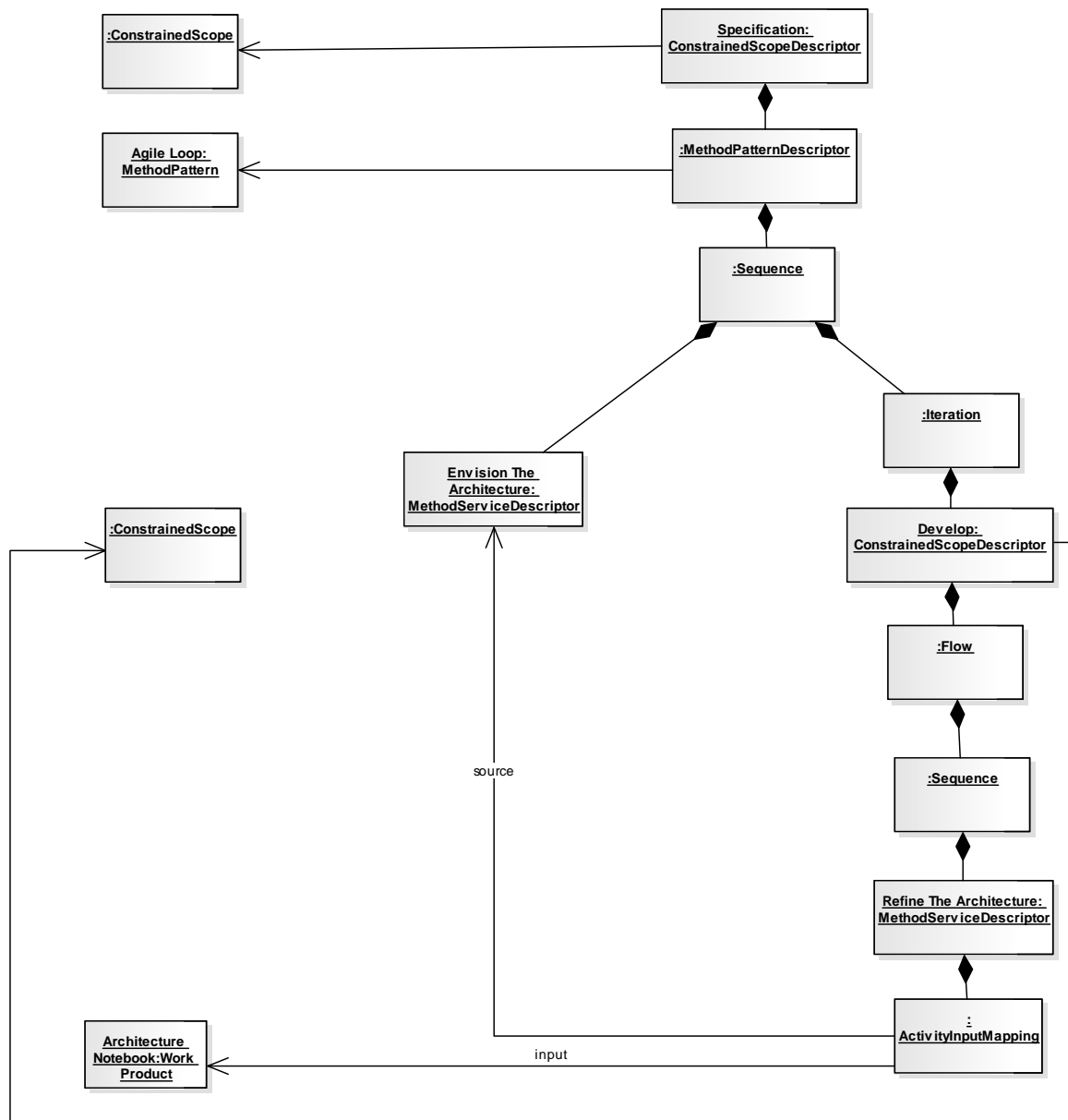


Fig. 5.6 The object diagram with the data flow specification between two method service descriptors according to Figure 3.22

(SPICE). Researchers discussed the difficulty and need of standardizing the notion of quality for methods already two decades ago [TRL96]. However, according to [Hen+14], in the last two major situational method engineering (SME) conferences ([RBH07],[RMD11]) only one paper on this topic was presented [ZSo7] in which the authors remarked that formal means to reason about method quality were still missing. Today, there is not even a standardized notion of method quality available like it exists for software product quality with the ISO/IEC 25010 [ISO11].

In [RB96], the authors propose to assess different methods based on their meta-models. They focus on the modeling of work products in different methods. Based on their meta-models (modeling languages), they propose different metrics based on the number of objects, attributes, and relationships to measure the complexity of a language.

In a similar manner, in his PhD thesis [Har97] and in a following publication [BSH98], Harmsen discusses five quality characteristics, e.g., completeness and consistency, in the context of method models composed with Demacrone. He formalizes them with first order logic based on the set-based formalization of methods used in Demacrone. We use this set of quality characteristics as a foundation for our quality model and formalization.

Following this line of thought, in [Chroo] the author describes a generalized meta-model for methods and postulates five well-formedness rules that should be fulfilled by method models, e.g., that for all work products there should exist a source (a producing activity) and a sink (a consuming activity). However, she does not categorize these rules with respect to a quality model.

In an attempt to enable the check of method models for their conformance to CMMI, the authors in [Hsu+08] propose a UML-based approach to define, verify, and validate method models. They define rules formalized with the Object Constraint Language (OCL) for their validation, e.g., to express that a WorkPractice (activity) is performed by exactly one ProcessPerformer (role). As our formalization of quality characteristics is also based on OCL, this work is related to ours. However, the focus in [Hsu+08] is on the conformance to CMMI of method models modeled with an UML-based approach and it does not offer a quality model for method models.

Related to the work in [Hsu+08] is eSPEM, using OCL in a SPEM-based approach. The authors of eSPEM claim that they implemented consistency rules based on OCL in their meta-model, however, details are not presented in the paper [Ell+11].

In [Per+11], the authors advocate the check of method models before enactment and propose well-formedness rules for SPEM in order to ensure the consistency of SPEM-based method models. On the one hand, the authors redefine the cardinality of some of the defined relationships of meta-classes, e.g., such that each TaskUse (activity) is associated to at least one ProcessPerformer (role). For more elaborated well-formedness rules the authors use first-order predicate logic, e.g., to ensure that work products are produced before they are consumed. The approach discussed in this paper is very closely related to our approach to formalize the quality of method models, but they do not discuss a possible implementation of their rules. In addition, the authors provide a list of low-level rules, but do not group or structure their well-formedness rules with respect to a quality model. Furthermore, their rules are limited to SPEM, which we had to extend due to its lack of executability.

In the following, we first present our quality model for method models that we derived based on the related work. We then present our concept for the automated quality analysis. Thereafter, we illustrate the usage of the automated quality analysis.

5.4.1 Quality Model

As explained, a quality model is a set of quality characteristics that can be further divided into sub-characteristics and eventually quality attributes, where quality attributes are measurable properties of an object [ISO01]. As we did not find a suitable quality model for the quality assurance of method models, we define our own quality model. In the following, we first present our quality model and then discuss it with respect to evaluation criteria for quality models.

The Quality Model of MESP

The quality model that we propose is inspired by the quality model of the ISO/IEC 9126 standard for software product quality [ISO01] and based on the related work in the field of SME. In the following, we describe and discuss the hierarchical structure of our quality model that is visualized in Figure 5.7. The general structure of our quality model is the following. The overall method model quality consists of three parts:

- Situation-independent and critical quality
- situation-independent and non-critical quality
- situation-dependent and non-critical quality

Each of these parts is refined into quality characteristics, which in turn are refined into quality sub-characteristics. Omitted in the figure are quality attributes. They are used to measure quality sub-characteristics.

Situation-Independent and Situation-Dependent Quality Quality characteristics can be classified based on whether their assessment is affected by the project situation. If they are, we call them *situation-dependent*. Otherwise, they are *situation-independent*.

Critical and Non-Critical Quality Quality characteristics can be classified based on their importance for the executability of the method model. If quality characteristics must be fully fulfilled in order to preserve the executability of a method model, we call them *critical*. Otherwise, we call them *non-critical*.

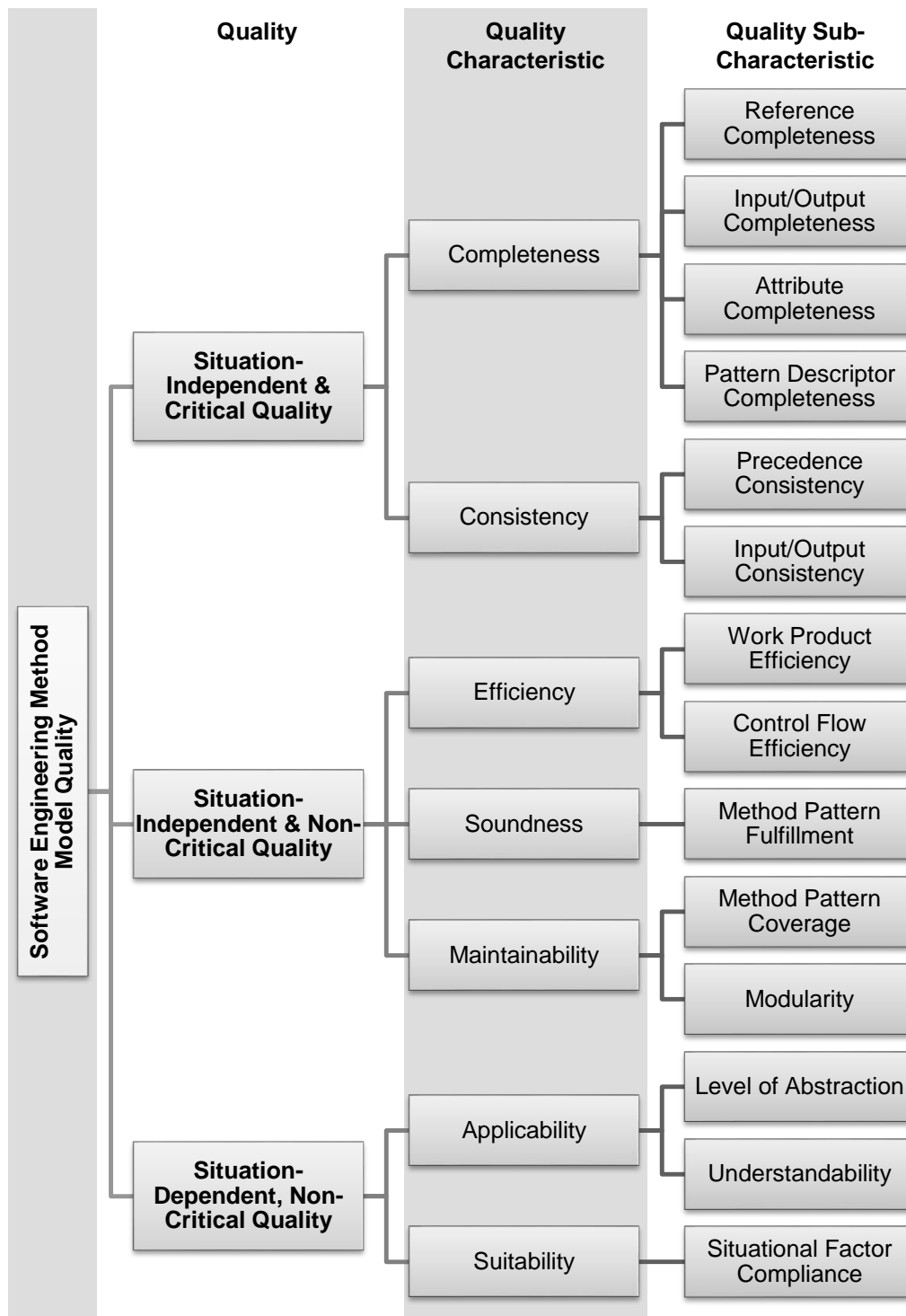


Fig. 5.7 MESP's quality model for software engineering method models

Completeness Completeness is fulfilled, if all required objects, attributes, and associations of a method model are defined. As missing elements can lead to cases, where the method model is not executable, independent of situational factor values, it is a situation-independent and critical quality characteristic.

We define completeness based on the quality characteristic with the same name defined in [Har97], where it requires that all method fragments referenced within a method model are existing and included.

Completeness is refined into the quality sub-characteristics *reference completeness*, *input/output completeness*, *attribute completeness*, and *method pattern completeness*. *Reference completeness* is fulfilled, if all mandatory associations as defined in the meta-model from objects contained within a method model to objects within the method repository exist. The meta-classes that can cause a violation of the reference completeness are task descriptors, method service descriptors, activity input mappings, method pattern descriptors, constrained scope descriptors, and project goals. Thus, the counts of their missing associations serve as quality attributes for reference completeness.

Input/output completeness is fulfilled, if the two following conditions are met. First, each activity within a method model that requires input work products contains the corresponding activity input mapping that specifies from where the input is coming. Second, for each output work product specified in the project goal of a method model the project goal must contain an activity output mapping that specifies from which activity the output is coming. Thus, the quality attributes are, first, the counts of missing activity input mappings for task descriptors and method service descriptors based on the mandatory input work products of the respective task or method service, and second, the counts of activity output mappings that are missing or that are missing mandatory associations based on the specified output work products in the project goal. As a stricter sub-characteristic of input/output completeness, we define the *conditional input/output completeness* that additionally takes into account, whether the source activity of the input work product is conditionally executed, thus might not produce the required input. This is the case, when the source activity is contained by an if, else if, or else activity *ca* and the consuming activity is not contained within the same *ca*. Conditional input/output completeness is fulfilled, if input/output completeness is fulfilled for all possible conditional control flows, ignoring activities that would not be executed.

Attribute completeness is fulfilled, if all mandatory attributes as defined in the meta-model are defined for objects contained within a method model. These mandatory attributes are the name of the project method, the name of an atomic activity, and the main description of a human decision. The count of missing mandatory attributes serves as quality attribute.

Pattern descriptor completeness is fulfilled, if each method pattern descriptor *mpd* in the method model has the same structure as the method pattern *mp* that it

references. This means that for every activity a in the containment hierarchy of mp there is a corresponding activity ca in the containment hierarchy of mpd . In addition, if an activity e has another activity ae in its containment hierarchy then the corresponding activity ca of e has to have a corresponding activity to ae in its containment hierarchy. A corresponding activity ce is an object of the same type that represents e and has the same attribute values. An exception are constrained scopes, where the corresponding activity for a constrained scope cs is a constrained scope descriptor that references cs . From pattern descriptor completeness follows method pattern consistency and vice versa. Thus, we did not list it as a separate quality sub-characteristic. It means that the structure of method pattern descriptors and thus the control flow containment relationship between constrained scope descriptors corresponds to the relationship given by the method pattern and its constrained scopes. The count of missing corresponding activities serves as quality attribute for pattern descriptor completeness.

Consistency Consistency is fulfilled, if the method model does not contain contradictions. Contradictions can lead to cases where the method model is not executable, independent of situational factor values; therefore, it is a situation-independent and critical quality characteristic.

We define consistency based on the quality characteristic with the same name defined in [Har97], where it requires that elements and their relationships within a method model do not contain any contradictions and thus are mutually consistent.

Consistency is refined into the quality sub-characteristics *precedence consistency* and *input/output consistency*. *Precedence consistency* is fulfilled, if for each data handling activity (cf. 5.3.2) that requires an input, the source of this input is a predecessor data handling activity or the input is already available initially at project start. This means that a source of an activity input mapping must be a predecessor, with respect to the control flow, of the data handling activity that contains the activity input mapping or it must be the data flow interface, hence the project goal. The quality attribute is the count of method service descriptors and task descriptors that have activity input mappings that do not fulfill this.

Input/output consistency is fulfilled, if the type of the data flow in the method model corresponds to the interface specification of tasks and method services in the method repository or the work products specified in the interface (project goal) of the method model. This means that for each activity input mapping ip with a referenced work product wp two conditions must be fulfilled. First, the data handling activity that contains ip has to reference a method service or task in the method repository that has wp among its input work products. Second, if the input provider that is referenced as source of ip is a data handling activity, it has to reference a method service or task in the method repository that has wp among its output work products, otherwise, it has to be a data flow interface (e.g.

project goal) that specifies the work product wp as an input available at project start. Similarly, for each activity output mapping op with a referenced work product wp the source data handling activity needs to reference a method service or task that has wp among its output work products and wp must be specified among the output work product of the data flow interface (project goal). The quality attribute for input/output consistency is the count of activity input mappings and activity output mappings that violate these conditions.

Efficiency Efficiency is fulfilled, if the method model fulfills its duty at minimal cost and effort. Missing efficiency, however, does not lead to cases where the method model is not executable and it is independent of situational factor values. Therefore, it is a situation-independent and non-critical quality characteristic.

We define efficiency based on the quality characteristic with the same name defined in [Har97], where it requires that activities that are not closely related in the method model do not produce the same output work products and that they do not receive input work products that are produced within that activity itself. As this definition is too strict, we define an adopted sub-characteristic. Efficiency is refined into the quality sub-characteristics *work product efficiency* and *control flow efficiency*.

Work product efficiency is fulfilled, if each output in the method model that is produced by a data handling activity (cf. 5.3.2) is used either as an input by another data handling activity or as an output of the method model (i.e. as a final result). This means that for each data handling activity dha and each output work product wp that is specified as output by the task or method service referenced by dha , there exist an activity input mapping with dha as a source and wp as the input work product or an activity output mapping with dha as a source and wp as the output work product. In addition, each work product that is specified as input for the method model by the project goal needs to be the source for an activity input mapping within the method model. The quality attribute for work product efficiency is the count of the missing activity input/output mappings according to the stated conditions.

Control Flow Efficiency is fulfilled, if the method model does not contain useless control flow constructs, which are flows, sequences, phases, iterations, ifs, else ifs, and elses that do not contain other elements (cf. 5.3.2). In addition, if-constructs, where the conditional (if) and the alternative (else) activities have the same content also violate this condition. The quality attribute for control flow efficiency is the count of useless control flow constructs.

Soundness Soundness is fulfilled, if the method model is semantically correct and meaningful. A method model can be unsound, but syntactically correct and

executable. Therefore, this is a situation-independent and non-critical quality characteristic.

We define soundness based on the quality characteristic with the same name and definition in [Harg7], but we define different sub-characteristics. For now, we only define the sub-characteristic *method pattern fulfillment*.

Method pattern fulfillment is given, if all used method patterns in a method model are fulfilled. This means, that for each constrained scope descriptor in the method model, the contents of it must fulfill the constraint described by the referenced constrained scope. The quality attribute for method pattern fulfillment is the number of unfulfilled constraints of constrained scope descriptors.

Maintainability Maintainability is fulfilled, if the method model can be easily adapted and managed. Missing maintainability, however, does not lead to cases where the method model is not executable and it is independent of situational factor values. Therefore, it is a situation-independent and non-critical quality characteristic. Maintainability is refined into the quality sub-characteristics *method pattern coverage* and *modularity*.

Method pattern coverage is given, if all method service descriptors and task descriptors of the method models are placed within method pattern descriptors, which eases meaningful changes and updates, as the used method patterns and their constraints cover the whole method model. This means that every method service descriptor and task descriptor is contained within the containment hierarchy of a constrained scope descriptor of a method pattern descriptor. The quality attribute for method pattern coverage is the number of method service descriptors and task descriptors that are not in the containment hierarchy of any constrained scope descriptor.

Modularity is given, if the method model is divided into manageable, well-defined units. In MESP, method models can reference method services and tasks. While the later are mainly textually described, atomic units of work, the former can wrap whole (sub-)processes that are described by a rich interface. Therefore, in method models the usage of method service descriptors instead of task descriptors is encouraged. In addition, regarding a group of method services (or tasks), the usage of a composite method service that includes them is encouraged over the usage of many single method services. The quality attributes for modularity are the number of method service descriptors, the number of task descriptors, and the number of tasks directly or indirectly referenced by the method model.

Applicability Applicability is fulfilled, if the method model is appropriate for the enactment by a project team. Thus, it is a situation-independent and non-critical quality characteristic.

Applicability is refined into the quality sub-characteristics *level of abstraction* and *understandability*. *Level of abstraction* describes on what level of detail the work to be performed is described. A low level of abstraction indicates that the method model describes the work to be performed in much detail, leaving less room and freedom for self-organization of the project team members, however, also requiring less qualification. On the contrary, a high level of abstraction indicates an abstract description of the work and more freedom for the project team members. The level of abstraction of a method model depends, on the one hand, on the number of tasks that it describes. On the other hand, it depends on the level of abstraction of each of these tasks. A task is less abstract, if it has a longer task description and less input and output work products, since this indicates a more detailed description of a smaller unit of work. Quality attributes for the level of abstraction are therefore the number of tasks directly or indirectly referenced by the method model as well as the length of the task description and the number of the input and output work products of each of these tasks.

Understandability is given, if the method model is understandable in terms of the readability of the referenced task descriptions and the complexity of the control and data flow. The easier the task descriptions are to read and the simpler the control and data flow are, the higher is the understandability. Quality attributes for the understandability are therefore the readability of the task descriptions, the number of data flows, and the number of conditional paths of control flow.

Suitability Suitability is fulfilled, if the process of the method model is appropriate for its project situation. As this quality characteristic does not affect the executability, it is a situation-dependent and non-critical quality characteristic.

Suitability is refined into the quality sub-characteristic *situational factor compliance*. *Situational factor compliance* is given, if the situational factor values in the project situation of a method model match the ones that characterize the method services referenced by it. It is violated in the following two cases. First, a situational factor is referenced by one or more of its situational factor values as an included factor of the project situation, and also by a method service referenced from the method model, however, no situational factor value matches. This signifies that a situational factor is relevant, but does not comply. Second, a method service referenced from the method model has a situational factor value that is among the excluded factors of the project situation.

Discussion of the Quality Model of MESP

In the previous section, we described a quality model for method models. A quality model should be of high quality itself and the question arises how to evaluate the quality model. In [GK09], the authors compiled a list of evaluation criteria for

quality models based on the existing literature on model quality. We list them in Table 5.2 (translated from German).

In the following, we discuss each of the evaluation criteria in the context of our quality model for MESP: Regarding *Clarity & Structuredness*, the quality characteristics of the MESP quality model are organized within a hierarchical tree structure providing a clear structure. Thus, this criterion is fulfilled. Regarding *Consistency*, as the quality characteristics of the MESP quality model are organized within a hierarchical tree structure, this criterion is also fulfilled. Regarding *Cohesion & Modularity*, the MESP quality model contains three layers of grouping and currently sixteen quality sub-characteristics. As all sub-characteristics belong semantically to their root characteristics, cohesion and modularity is provided. Regarding *Unambiguity & Comprehensibility*, the quality characteristics of the MESP quality model are described with natural language in order to enhance the comprehensibility and as a foundation for additional quality sub-characteristics. As natural language is ambiguous, ultimately, quality sub-characteristics should be formalized with a formal language. For some quality sub-characteristics, we discuss the formalization in Section 5.4.2. However, formalizing all quality characteristics is still an open research issue and beyond the scope of this thesis. Overall, we consider this criterion as fulfilled. Regarding *Absence of Overlappings*, the quality characteristics of the MESP quality model are free from redundancies. For example, as from pattern descriptor completeness follows pattern descriptor consistency and vice versa, we omitted the latter form the quality model. Thus, this criterion is fulfilled. Regarding *Operationalizability*, all quality characteristics of the MESP quality model are operationalizable and we either stated the quality attributes to measure or provide the operationalization with OCL as part of our solution. However, as described with respect to the unambiguity, the formalization of all quality sub-characteristics is beyond the scope of this thesis. Regarding *Completeness & Relevance*, our MESP quality model is intended to support project method engineers in assuring the quality of composed method models. In this regard, all presented quality characteristics are relevant as they influence the perceived quality of the method model. Regarding the completeness, this quality model is based on the related work in the field. However, we do not claim that the quality model is complete as further, especially non-critical, quality sub-characteristics might be proposed in the future. With respect to the critical quality characteristics, we consider it complete. Regarding *Correctness and Consistency*, the MESP quality model conforms to the discussed structure. Quality characteristics, especially quality sub-characteristics, and quality attributes are discussed with respect to the MESP meta-model and method models. Therefore, we consider this criterion fulfilled. Regarding *Satisfiability*, the quality characteristics of the MESP quality model are discussed with respect to structural features of method models. Hence, the underlying quality goals are realistic and satisfiable.

Table 5.2 Evaluation Criteria for quality models (adopted from [GK09])

<i>Evaluation Criterion</i>	Description
<i>Clarity & Structuredness</i>	models need to be clear, readable, and structured as a hierarchy.
<i>Consistency</i>	quality characteristics need to be transitive, thus the quality model shall not contain two quality characteristics x and y , where both x is a (possibly indirect) sub-characteristic of y and y is a (possibly indirect) sub-characteristic of x .
<i>Cohesion & Modularity</i>	quality characteristics that belong together, should also be closely connected in the quality model. In addition, each quality characteristic of the quality model should represent only one aspect of quality.
<i>Unambiguity & Comprehensibility</i>	Every quality characteristic shall be clearly and comprehensibly described.
<i>Absence of Overlappings</i>	Quality characteristics must not be overlapping and should be pairwise disjoint and free of redundancies.
<i>Operationalizability</i>	All quality characteristics should be measurable (by quality attributes). If a subjective assessment is required, the quality characteristic should be labeled as such.
<i>Completeness & Relevance</i>	A quality model should comprise all quality characteristics and all their relationships that are relevant for its stakeholders, such that its value decreases, when quality characteristics and relationships are removed.
<i>Correctness & Consistency</i>	A quality model should be consistent to its meta-model. Additionally, its quality characteristics and quality attributes should conform to the nature of the domain.
<i>Satisfiability</i>	The quality goals expressed with the quality model and its quality characteristics should be realistic and satisfiable by the models.

After we discussed the underlying quality model, in the following section, we discuss the concept of our automated quality assurance framework.

5.4.2 Automated Quality Assurance Framework

In this section, we describe the concept of our automated quality assurance framework that supports the project method engineer. In the following, we first present the requirements for such a framework. Thereafter, we give a conceptual overview of it. Then, we discuss how we formalized quality characteristics of the MESP quality model, in order to allow for automated assessment of the method models.

Requirements

For this analysis framework, we define the requirements listed in Table 5.3 as a refinement of MTR5. In order to provide early feedback, the analysis shall be applicable to incomplete method models and restrictable to single regions (activities, see Section 5.3.2) of the model (MTR5.1). In order to help in fixing quality issues, the elements of a method model that cause quality issues shall be reported (MTR5.2). In order to help in assessing quality issues, the analysis shall classify quality issues based on their type (regarding a quality model). Most importantly, critical and non-critical issues shall be distinguished, where critical issues prevent method models from being enacted with a process engine (MTR5.3). In order to implement further quality checks easily, the analysis framework shall be extensible (MTR5.4). And finally, in order to be used frequently throughout the composition of a method model, the analysis shall not take longer than a couple of seconds (MTR5.5).

Table 5.3 Refined requirements for the analysis framework

<i>Requirement</i>	Name
<i>MTR5.1</i>	Partial analysis of models
<i>MTR5.2</i>	Traceability of issues
<i>MTR5.3</i>	Categorization of issues
<i>MTR5.4</i>	Extensibility of the analysis framework
<i>MTR5.5</i>	High performance of the analysis

Overview of the Quality Assurance Framework

The MESP tool support is based on the Eclipse Modeling Framework [Ste+09] that allows automatically checking for the conformance of models with their meta-models. This allows analyzing method models for reference completeness and

attribute completeness (see Section 5.4.1) out of the box. However, the fulfillment of other quality characteristics cannot be checked with the build-in support. Therefore, we propose an extensible quality assurance framework that is implemented as the consistency checker component in the tool support of MESP (see Figure 3.4).

The concept of the quality assurance framework is shown in Figure 5.8. The automated quality analysis is based on the formalization of quality characteristics with the OCL [OMG14]. The analysis consists of two parts. First, the method model is evaluated against a set of pre-formalized quality constraints (1.). These constraints are represented by a set of pre-defined OCL expressions. This part is discussed in Section 5.4.2. Second, the method model is evaluated against the method pattern constraints used in the method model (2.). As method patterns can be added or modified, we cannot provide a pre-defined set of equivalent OCL expressions. Instead, method pattern constraints are translated on the fly to OCL and evaluated against the model. This part is discussed in Section 5.4.2. For the evaluation of OCL expressions, our framework reuses the existing Eclipse OCL Component. The detected issues and pattern violations are passed, together with the responsible model elements, to the Eclipse Problems View component and are then presented to the project method engineer.

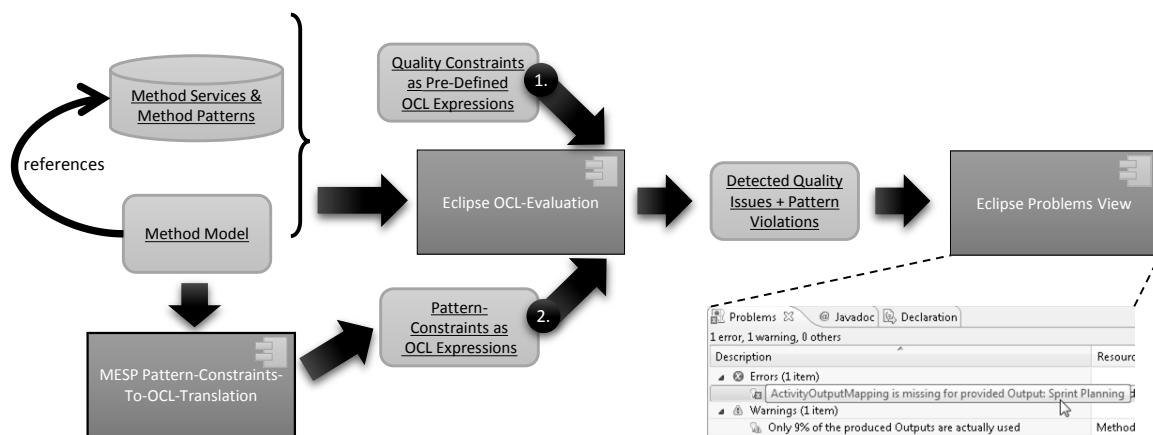


Fig. 5.8 Overview of the quality assurance framework

Formalization and Analysis of General Quality Characteristics

Most quality characteristics can be formalized upfront. This set of pre-defined OCL expressions can then be used to automatically analyze method models. Every time, the user invokes the quality analysis, the method model is checked against all pre-defined OCL expressions and the quality issues are reported. We have currently implemented a set of six critical and two non-critical quality sub-characteristics (c.f. requirement MTR5.3) to demonstrate the approach and the extensibility of the framework (c.f. requirement MTR5.4) and to cover all essential aspects to ensure

the enactment of method models. Violations of critical quality sub-characteristics are presented as errors, while violations of non-critical sub-characteristics are presented as warnings. Thus, the project method engineer is getting feedback on what issues must be resolved in order to derive an executable method model. The implementation of further non-critical quality characteristics remains for future work, especially, as further quality sub-characteristics might be proposed in the future.

We exemplify the derivation of OCL expressions using precedence consistency as an example (see Section 5.4.1). Figure 5.9 shows the OCL constraint that formalizes this quality sub-characteristic. It computes the set of all method service descriptors or task descriptors of the process (lines 1-4) and removes from this set (line 5) all descriptors that have only valid sources as the origin of their input work products. Valid sources means that the origin input provider of the input work product is a predecessor method service descriptor or predecessor task descriptor (lines 5-7). The functions `getNested()` and `getPredecessors()` are OCL helper functions that we defined. The first returns the set of all contained activities, while the latter returns all activities that, regarding control flow, are executed before. If the resulting set computed by the expression is not empty, it contains the method service descriptors and task descriptors that violate the quality characteristic (c.f. requirement MTR5.2).

```

1 context Process
2 self.activity.getNested()->
3 select(oclIsTypeOf(MethodServiceDescriptor) or
4       oclIsTypeOf(TaskDescriptor))->
5   reject(prov | prov.inputs.source->
6         forALL(sourceActivity |
7             activity.getPredecessors(prov)->includes(sourceActivity)
8         )

```

Fig. 5.9 OCL expression to find elements that violate precedence consistency

As another example for the formalization with OCL, Figure 5.10 shows the definition of the OCL helper function `getNested()` used in the OCL constraint for the precedence consistency. It basically differentiates three cases in order to collect the activities in the containment hierarchy of an activity, ignoring e.g. activity input mappings. Case 1 is given, if the function is invoked upon an activity with activities (line 3-11). Case 2 is given, if it is invoked upon a task descriptor or method service descriptor. Case 3 is given, if it is invoked upon another type of activity. In case 1, the function returns any directly contained task descriptors and method service descriptors (line 5-6). Additionally, for all other directly contained elements the contained activities are returned using a recursive call to `getNested()` (line 9). In case 2, the function returns the task descriptor or method service descriptor,

respectively (line 14-15). In case 3, the function returns the contained activities of the element using a recursive call to `getNested()` (line 19). In addition, in all cases, the element itself (`self`) is returned (line 11 and line 23).

```

1  getNested():Set(Activity) =
2  if self.oclIsKindOf(ActivityWithActivities) then
3    self.activities->iterate(
4      a:Activity;result:Set(Activity) = Set{} |
5      if( a.oclIsTypeOf(TaskDescriptor) or
6        a.oclIsTypeOf(MethodServiceDescriptor) ) then
7        result->including(a)
8      else
9        result->union(a.getNested())
10     endif
11   )->including(self)
12 else (
13   if self.activity->notEmpty() and
14     (self.activity.oclIsTypeOf(TaskDescriptor) or
15     self.activity.oclIsTypeOf(MethodServiceDescriptor)) then
16     Set{}->including(self.activity)
17   else
18     if self.activity->notEmpty() then
19       Set{}->union(self.activity.getNested())
20     else Set{}
21     endif
22   endif
23 )->including(self)
24 endif

```

Fig. 5.10 OCL definition of the helper function `getNested()`

Formalization and Analysis of Method Pattern Constraints

For the quality sub-characteristic method pattern fulfillment, the method pattern constraints of the constrained scope descriptors have to be evaluated. Unlike other quality constraints, the method pattern fulfillment cannot be formalized upfront with pre-defined OCL expressions. Because senior method engineers can define arbitrary method pattern constraints for their method patterns, there is no fixed set of method pattern constraints. For this reason, we use an on-the-fly translation for method pattern constraints into equivalent OCL expressions (see Figure 5.8), so that we can reuse the existing Eclipse OCL Component to evaluate method pattern constraints against the process model and determine method pattern fulfillment.

The translation works as follows (see Figure 5.8): Based on the used method pattern descriptors in the method model, the corresponding constraints from the method repository are extracted and transformed into equivalent OCL expressions. Then these are checked against the respective constrained scope descriptors of the method model. These pattern-related OCL expressions are created on-the-fly

during each run of the quality assurance analysis. Furthermore, method pattern constraints are evaluated only against the respective constrained scope descriptor, while the other quality constraints are evaluated against all elements in the scope of the analysis, so usually the whole method model.

Figure 5.11 shows a Java code snippet used in the construction of the OCL expression for a method pattern constraint. The depicted code adds the part of the OCL expression that expresses the used quantifier of a single condition element (see Section 4.4.2). The quantifier states, whether the method pattern constraint has to be fulfilled for all, for none, or at least one atomic activity within the constraint scope descriptor. In the first case (line 3-10), the number of the elements that fulfill the condition (line 6) has to match the number of the method service descriptors and task descriptors in the containment hierarchy of the constrained scope (line 6-9). In the second case (line 11-15), the set of elements that fulfill the condition shall not be empty (line 14). In the last case (line 16-19), the set of elements that fulfill the condition shall be empty (line 18). The code in the lines 19-21 is part of the error handling.

```
1 private String parseSingleCondition(SingleCondition sc)
2 {
3     if(sc.getExpression().getQuantifier() ==
4         Quantifier.ALL_FULFILL)
5     {
6         conditionString+= "->size() = "+
7             "self.getNested()->"+
8             "select(oclIsTypeOf(MethodServiceDescriptor))+
9             "->size()";
10    }
11    else if (sc.getExpression().getQuantifier() ==
12        Quantifier.AT_LEAST_ONE_FULFILLS)
13    {
14        conditionString+= "->notEmpty()";
15    }
16    else if (sc.getExpression().getQuantifier() ==
17        Quantifier.NONE_FULFILLS) {
18        conditionString+="->isEmpty()";
19    } else
20    { return unableToParse; //Unknown Condition
21    }
22    return conditionString;
23 }
```

Fig. 5.11 A Java code snippet for the translation of quantifiers to OCL

5.4.3 Usage

With our automated quality assurance framework, we enable the automated quality analysis of method models. This helps project method engineers in composing consistent method models. In the following, we discuss two examples, one for the check against general quality characteristics formalized upfront and one for the check against method pattern constraints transformed to OCL on-the-fly.

We use again precedence consistency as an example for general quality characteristics. Figure 5.12 shows a derivation of Figure 3.22 from the end-to-end example in Chapter 3. The method service descriptor “Refine the Architecture” gets its input work product from “Envision the Architecture”, however, that is executed afterward and hence cannot produce the input work product. Thus, the quality sub-characteristic precedence consistency is violated. In this case, the evaluation of the OCL expression presented in Figure 5.9 returns “Refine the Architecture” as violating element as indicated by the \otimes next to the method service descriptor. In the consistency checker component, a reference to this element is passed along with an error description to the Eclipse Problems View component such that it is presented to the project method engineer. She can then fix the issue, e.g., by changing the control flow order of the elements as discussed in Chapter 3 (cf. 3.22).

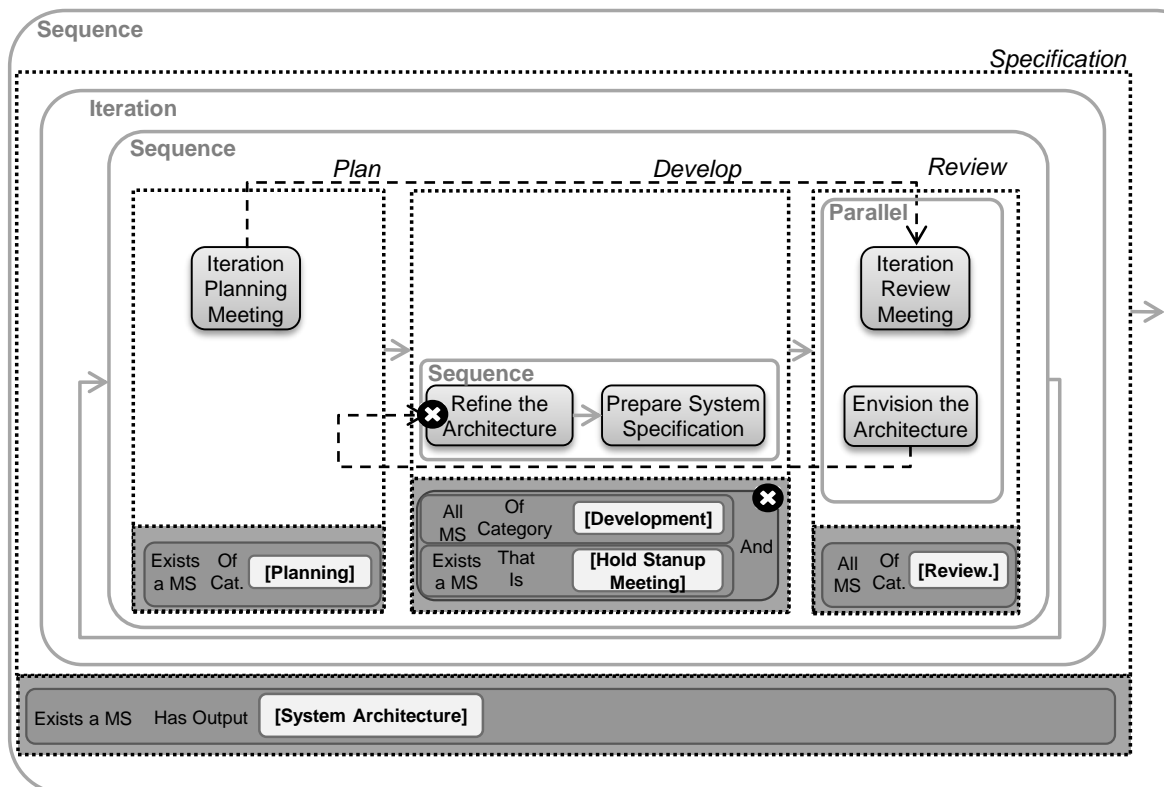


Fig. 5.12 A partial method model with quality issues

Also shown in Figure 5.12 is a violation of the method pattern sub-characteristic. The method pattern constraint of the middle constraint scope descriptor is not fulfilled as indicated by the \otimes in the upper right corner of the constraint. With the assumption that both method service descriptors reference method services that have the “Development” category, the upper condition is fulfilled, however, as there is no “Hold Standup Meeting” method service referenced in the constrained scope descriptor, the lower condition is not fulfilled. Thus, the overall evaluation result is also negative.

This violation is detected by evaluating the OCL expressions presented in Figure 5.13 that are generated on-the-fly. The equivalent OCL expression for the upper condition is shown in lines 1-11. The equivalent OCL expressions for the lower condition is shown in lines 13-20. In accordance with the upper condition, the OCL expression collects the method service descriptors that have the category “Development” (line 4-8). The size of the set with these elements is then compared to the size of the set with all method service descriptors within the constrained scope descriptor (line 9-11). These lines are generated by the lines 3-10 of the Java code shown in Figure 5.11. For the lower condition, the OCL expression collects the method service descriptors with the name “Hold Standup Meeting” (lines 15-17) and checks whether the set with these elements is empty (line 20). This lines are generated by the line 11-15 of the Java code shown in Figure 5.11.

```

1  // Upper Condition
2  context ConstrainedScopeDescriptor
3  getNested()->
4  select(oclIsTypeOf(MethodServiceDescriptor))->
5  select(msd | msd.methodService.interface.categories->
6    exist(category | category.name = 'Development'))->
7  iterate(x:MethodServiceDescriptor; activitySet:Set(Activity) =
8    Set{} | activitySet->including(x) )
9  ->size() =
10 self.getNested()->
11 select(oclIsTypeOf(MethodServiceDescriptor))->size()
12
13 // Lower Condition
14 context ConstrainedScopeDescriptor
15 getNested()->
16 select(oclIsTypeOf(MethodServiceDescriptor))->
17 select(msd | msd.methodService.name = 'Hold Standup Meeting')->
18 iterate(x:MethodServiceDescriptor; activitySet:Set(Activity) =
19   Set{} | activitySet->including(x) )
20 ->nonEmpty()

```

Fig. 5.13 The generated OCL expressions for the middle constraint scope descriptor of Figure 5.12

In the consistency checker component, the two OCL expressions are evaluated against the respective constrained scope descriptor using the Eclipse OCL Com-

ponent. The result of the overall `AndCondition` is then computed using a Java “AND”-Statement. As this evaluates to false, a reference to the middle constrained scope descriptor is passed along with an error description to the Eclipse Problems View component such that it is presented to the project method engineer. She can then fix the issue, by adding a method service descriptor that references the method service “Hold Standup Meeting” as shown in the end-to-end example in Chapter 3 (cf. Figure 3.22). Similar to the discussed example, all possible method pattern constraints can be transformed to OCL and consequently evaluated with our quality analysis.

After we illustrated the quality analysis, we revisit the requirements described in Section 5.4.2 and discuss their fulfillment. Regarding requirement MTR5.1, our implementation of the automated quality assurance framework can be invoked on the whole Process, but also on single Activities. It is designed to run also on method models that are only partially completed. Thus, this requirement is fulfilled. Regarding requirement MTR5.2, our formalization of quality characteristic collects model elements that cause quality issues (cf. Section 5.4.2). Quality issues are presented to the project method engineer in the Eclipse Problems View (cf. Figure 5.8). If the user double-clicks on an issue in the Problems View of our MESP tool support, she jumps to the causing element. In addition, the elements that cause quality issues are highlighted in the Method Composer and Repository Browser. Thus, this requirement is fulfilled. Regarding requirement MTR5.3, critical issues are presented as errors and non-critical issues as warnings in the Eclipse Problems View (cf. Section 5.4.2). In addition, the issues are categorized according to our quality model (cf. Section 5.4.1) in their description. Therefore, this requirement is fulfilled. Regarding the requirement MTR5.4, our automated quality assurance framework is not limited to a specific set of quality sub-characteristics, because additional quality sub-characteristics can always be added by extending the set of OCL expressions. In Section 5.4.2, we explained the set of quality sub-characteristics that we have already implemented using this mechanism. Thus, this requirement is fulfilled. Regarding the last requirement MTR5.5, we evaluated the runtime of the quality analysis (cf. discussion in Chapter 7). As the evaluation showed, the analysis is fast enough for realistically sized method models, thus this requirement is also fulfilled. In summary, our quality assurance framework fulfills all the requirements that we derived.

5.5 Initialize Method

In this section, we discuss how the project method engineer initializes the method model in order to prepare it for execution. Initializing the method model comprises two tasks. First, the method model needs to be transformed into a process model that can be executed with a standard process engine. Second, this process model

needs to be deployed on the process engine. The process engine and the standard project repository for work products need to be configured such that they can be used by the project team. We offer an automated transformation and deployment (MTR6) as part of our tool support as published in [FCE14]. It is in part based on the results of a master thesis [Nee14].

Regarding the related work, there are three groups of approaches that are related. The first group of related work are approaches that do not directly support the execution of the method model, but support the enactment by providing documentation and tools. For example, tools like the EPF Composer or IBM Rational Method Composer allow defining SPEM-based method models. From these method models, a website can be generated that allows team members to browse for the description of processes and tasks (cf. Figure 3.9). The configuration-based approach MC Sandbox [KÅ11; KÅ12] offers similar support. The tool support of the V-Modell XT allows deriving project plans and document templates for work products used in the method model [KTF11]. Other approaches allow deriving a CASE environment from the method models, e.g., MERU [PS97] or *Method Management Tools* [KLR96]. An approach with this capability based on SPEM and modern technologies is MOSKitt4ME [Cer+11]. Based on the method model tools are automatically bundled into a CASE environment to use during method enactment. However, the approach has no process execution support as execution semantics are missing in SPEM.

Another group of related work are ALM suites [KV09] like Microsoft Team Foundation Server (TFS) or IBM Rational Team Concert. As explained in Section 2.2.2, these offer limited execution support based on automated workflows that can be adapted manually. However, the creation of situational methods is not supported explicitly.

The last group of related work are approaches that allow executing the method model with means of a process engine. The tool Demacrone of Brinkkemper and Harmsen allows generating a process engine based on the composed method model [Har97]. The Guidance Engine of MENTOR [SRG96; Pli96] also allows executing the selected method building blocks in order to offer guidance to the application developers. However, both approaches are based on low-level modeling languages and outdated technologies. A more standard-conforming way to allow for execution support is to follow our approach and to extend SPEM in order to overcome its lack of executability. Bendraou et al. [Ben+07] propose the extensions called xSPEM, but only sketch some mappings between concepts of xSPEM and BPEL and thus misses out on explicit support for human interaction. However, the approach follows the same idea that we implemented. Ellner et al. [Ell+10; Ell+11] propose the extension eSPEM and support the execution of methods based on an own process engine for eSPEM. However, there is no explicit support for situational method engineering.

In the following, we first present the transformation of method models into BPEL process models. Then we discuss their deployment and configuration for enactment by the project team.

5.5.1 Transformation, Deployment & Configuration

In order to execute method models an execution environment is required. If this execution environment, a process engine, does not exist, there are two possibilities. First, one could implement a process engine that directly supports the software engineering method modeling language, similar to the approach used for eSPeM. Second, the software engineering method modeling language can be mapped to another executable language, which is accompanied by process execution support. This approach allows reusing proven and stable execution support. In our MESP approach, we use this approach and transform our method models to BPEL/BPEL4People process models. In the following, we first explain how the execution of BPEL/BPEL4People processes works from a technical perspective. Thereafter, we present our mapping of language constructs and discuss the transformation of MESP method models to BPEL/BPEL4People process models. Thereafter, we explain the deployment and configuration and then illustrate the initialization of method models.

Process Enactment from a Technical Perspective

To execute a MESP method model, it is transformed into a BPEL/BPEL4People process model. BPEL is a executable process description language to coordinate web services and the BPEL4People extension enables to incorporate human interaction into BPEL processes. Thus, every BPEL/BPEL4People process model basically consists of two parts that are deployed into the BPEL engine: a BPEL process that is responsible for the general control and data flow and BPEL4People HumanTask services that creates workflow tasks and provides their graphical user interfaces (GUI) to interact with the project team members. From the perspective of the BPEL process, a HumanTask is just a regular web service that can be invoked with certain parameters and that returns a result. While the BPEL process and the HumanTask service reside inside the process engine component, the created workflow tasks are managed by the task management component of the BPEL engine. Figure 5.14 shows the interplay between the different parts. A BPEL4People `peopleActivity` within the BPEL process can invoke a HumanTask web service during process execution (1). This invocation then triggers the creation of a workflow task (2) for a project team member in the task management component of the BPEL engine. When the respective project team member selects a workflow task to enact it, she is presented the GUI defined by the respective HumanTask web service (3). Once the team member has provided her input and is finished (4), the result is replied back

to the BPEL process (5) and the next (potentially ordinary) service invocation can take place (7-9).

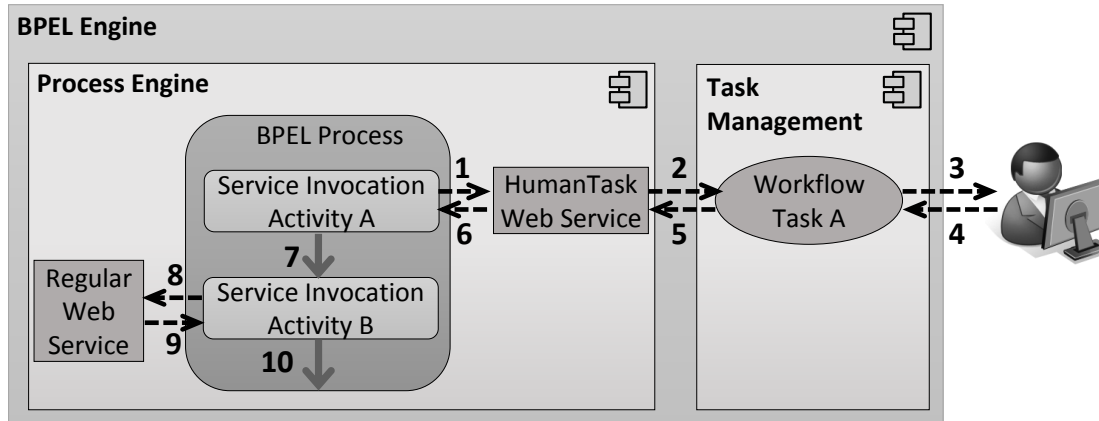


Fig. 5.14 The execution of a BPEL/BPEL4People process model

Mapping and Transformation to BPEL/BPEL4People

After we described the BPEL/BPEL4People process enactment, we now discuss the mapping between concepts of MESP and BPEL/BPEL4People. A straight forward way to define a mapping between MESP and BPEL/BPEL4People would be to create a dedicated HumanTask web service and its GUI for each task or decision of a method model. Each TaskDescriptor would then be transformed to an invocation of the appropriate HumanTask web service. The created workflow task could then be processed via the provided GUI that would contain the hard-coded information of the task, e.g., its description. However, this would result in creating, deploying, and maintaining many HumanTask web services together with their specific GUIs.

With MESP, we follow a different approach that reduces the number of HumanTask web services and GUIs required. We use a single, generic HumanTask web service and dynamic GUIs for all the Tasks referenced in a method model. The task-specific information, e.g., the task description or responsible role, is then encoded in the input parameters of the service invocation and loaded into the dynamic GUI. Technically, all Tasks are transformed into the same type of workflow task and TaskDescriptors can be transformed into invocations with appropriate parameters of the generic HumanTask web service.

Beside the type of workflow task that represents MESP Tasks (1), there are two further types of workflow tasks that are needed. The next type of workflow task represents MESP HumanDecisions (2) and allows deciding about conditional activities, e.g., another execution of an Iteration or executing If or Else activities. The third type of workflow tasks lets the project team set up the initial input work products of the method model according to the ProjectGoal. As these three

different types of workflow task differ in their input and output parameters an individual BPEL4People HumanTask web service is used for each type. Thus, for each MESP method model, we generate and deploy a BPEL process and three HumanTask web services. All tasks and decisions to be processed by the project team are transformed into invocations of one of these three HumanTask web services.

After we explained the general approach of our transformation, we provide an overview of the mapping between concepts of MESP and BPEL/BPEL4People in Table 5.4 and briefly describe the transformation of each MESP concept in the following.

Table 5.4 The mapping of MESP and BPEL/BPEL4People concepts

<i>MESP Concept</i>	<i>BPEL/BPEL4People Concept</i>
Process	process with global <variable> elements to store input and output work product values and HumanTask-related <peopleActivity> to initialize input WorkProducts
Sequence	<sequence>
Flow	<flow>
Phase	hard-coded <variable> for invocation of TaskDescriptor-related <peopleActivity>
Iteration	<repeatUntil> loop containing counting <variable>, HumanTask-related <peopleActivity> for looping condition, <assign> to increment counting <variable>
If, Else	<if> activity and HumanTask-related <peopleActivity> for condition
MethodPatternDescriptor	no explicit element
ConstrainedScopeDescriptor	no explicit element
MethodServiceDescriptor	no explicit element
TaskDescriptor	<assign> activity to initialize the input <variable>, <peopleActivity>, and <assign> activity to save the return values

Process The Process element is transformed into a BPEL <process> that contains <variable> declarations for all output WorkProducts of all TaskDescriptors directly and indirectly referenced from the method model. Additionally,

it contains `<variable>` declarations for all `WorkProducts` available at project start. Indirectly referenced `TaskDescriptors` are those that are used in `MethodServices` that are referenced by the method model using `MethodServiceDescriptors`. Here, several levels of indirection are possible, when `MethodServices` themselves contain `MethodServiceDescriptors`. Additionally, as the very first activity of the contained `<sequence>`, it contains a `HumanTask` invocation (`<peopleActivity>`) that is used to initialize the input `WorkProducts` available at project start. It creates a workflow task that allows the project team to specify the location of the respective `WorkProducts`.

Sequence, Flow These control flow activities can directly be mapped to their BPEL counterparts as they were derived from there.

Phase The `Phase` element is transformed into a textual hint for the workflow tasks of that `Phase`. Therefore, its name is provided as an invocation parameter `<variable>` to all of the `HumanTask` invocations (`<peopleActivity>`) that are contained in the `Phase`, such that it can be presented to the project team member (cf. Figure 3.27).

Iteration The `Iteration` element is transformed into a conditional `<repeatUntil>` loop that contains the contents of the `Iteration`. In order to allow the project team to decide about a further execution of the loop, a `HumanTask` invocation (`<peopleActivity>`) is added at the end of the loop (technically a further `<sequence>` element is created, where the transformed content of the loop is the first element and the `<peopleActivity>` the second). It represents the `HumanDecision` and creates a corresponding workflow task. Its return value (the human decision) is evaluated in the `<condition>` of the `<repeatUntil>` activity and based on its value, the loop is repeated. In addition, a BPEL `<variable>` is incremented at the end of the loop with an `<assign>` activity to count the number of iteration runs. Its value is shown as a textual hint in the workflow tasks of the loop. Therefore, it is provided as an invocation parameter `<variable>` to all of the `HumanTask` invocations (`<peopleActivity>`) that are contained in the loop, such that it can be presented to the project team member (cf. Figure 3.27).

If, Else The `If` element is transformed into a conditional `<if>` activity that contains the contents of the MESP `If` activity. A possibly existing `Else` activity is transformed into the `<else>` branch of the `<if>` activity. In order to allow the project team to decide about the execution of the `If` activity, a `HumanTask` invocation (`<peopleActivity>`) is added right before the `<if>` element (technically a further `<sequence>` element is created, where the transformed `If` activity is the first element and the `<peopleActivity>` the second). It represents the `HumanDecision` and creates a corresponding workflow task.

Its return value (the human decision) is evaluated in the <condition> of the <if> activity.

MethodPatternDescriptor It is not directly transformed into BPEL, but the contents of the MethodPatternDescriptor are recursively transformed and added to the BPEL <process>.

ConstrainedScopeDescriptor It is not directly transformed into BPEL, but the contents are recursively transformed and added to the BPEL <process>.

MethodServiceDescriptor It is not directly transformed into BPEL, but the contents of the referenced MethodService are recursively transformed and added to the BPEL <process>.

TaskDescriptor A TaskDescriptor is transformed into the invocation of a HumanTask (<peopleActivity>) that will trigger the creation of a workflow task. To do this the invocation parameter <variable> has to be initialized with an <assign> activity, so that the right information is presented to the project team member. First, a part of the invocation parameter <variable> is assigned values from the referenced MESP Task, e.g., its name, its description, and the associated Role. Second, a part of the invocation parameter <variable> is assigned values from the Process, e.g., the current Phase and the current Iteration. Third, a part of the invocation parameter <variable> is assigned values of WorkProduct <variable> elements according to the specified InputMapping of the TaskDescriptor, e.g., the value “http://redmine.s-lab.de/issues/142” with respect to Figure 3.27. After these <assign> activities the HumanTask can be invoked with the initialized invocation parameter <variable> using a <peopleActivity>. Afterward, another <assign> activity saves the provided outputs by the project team member into a work product <variable> for that output.

We implemented an algorithm that performs the described transformations automatically and generates the process model related files such that they can be deployed on the BPEL engine. The algorithm processes the process model in three steps. First, it replaces all usages of MethodServiceDescriptors with the contents of the Process of the referenced MethodService. As the replacing Sequence might itself contain MethodServiceDescriptors, it repeats this procedure until the process model does not contain any MethodServiceDescriptors anymore. In the second step, the algorithm builds up the BPEL process by traversing the Process of the method model and transforming the elements according to the description. In the third step, the algorithm creates the three HumanTask web service description files (cf. previous section). These three service descriptions are basically the same for all generated process models and are only configured with

different names and namespaces according to the respective method model for technical reasons.

Deployment & Configuration

Once the process model is derived by transforming the process model into the BPEL process and the three HumanTask web service description files (cf. previous section), it can be deployed and configured. Typically, deployment to a process engine is done by moving the process model files into a dedicated folder or by uploading them using a web front end. Our tool support allows deploying the files into the used BPEL engine without human interaction.

Thereafter, the process model needs to be prepared for execution and the project repository needs to be set up. On the one hand, the roles used in the method model need to be assigned to project team members. This requires the creation of the respective user accounts in both systems. On the other hand, the process model needs to be initialized with the values of the work products that are available at project start. To do this, the project method engineer first uploads the available work products to the project repository. Then she starts the execution of the process model using the configuration interface of the BPEL engine. This executes the initialization `<peopleActivity>` that creates a workflow task for the initialization with input work products of the project. Here, the project manager can enter the location URIs of the work products that were uploaded to the project repository.

5.5.2 Usage

With our automated transformation of method models into standard BPEL/BPEL-4People process models, we enable the execution of method models with a standard-conformant BPEL engine later by the project team. Figure 5.15 shows a simplified excerpt of the BPEL process that is transformed from the MESP method model discussed in the end-to-end example in Section 3.2. Lines 3-4 show the generated BPEL `<variable>` declarations for all output WorkProducts of all TaskDescriptors. Lines 8-32 show the transformed TaskDescriptor for the Task “Refine the Architecture”. The first part, in lines 8-18, shows the assignment (`<assign>` activities) of `<variables>` to the input parameter `<variable>` for the HumanTask invocation. Lines 11-14 show assignments for the Task-related name and description. Lines 15-17 show the assignment of the input WorkProduct URL value that is the output of the Task “Envision the Architecture”. The second part, in lines 20-24, shows the `<peopleActivity>` that invokes the HumanTask with the input `<variable>`. The last part, in lines 26-32, shows how after execution of the `<peopleActivity>` the output is saved. It is saved with an `<assign>` activity into the according output `<variable>` declared in line 4. The remaining parts of the method model are transformed in a similar manner.


```

1  <process name="ePassportMethod"> ...
2  <!-- variable declarations for all outputs of all task descriptors-->
3  <variable name="EnvisionTheArchitecture_architecture_notebook_OP"/>
4  <variable name="RefineTheArchitecture_architecture_notebook_OP"/>
5  ...
6  <sequence>
7  <!-- parameter assignments to set up the invocation of HumanTask-->
8  <assign name="RefineTheArchitecture_Input">
9  <!-- copy values from MESP task to invocation parameter -->
10 <copy> <from> ...
11 <taskName>RefineTheArchitecture</taskName>
12 <descr>"This task builds upon the outlined architect..." </descr>
13 ...
14 </from> <to variable="b4pInput"> </to> </copy>
15 <copy> <from> $EnvisionTheArchitecture_architecture_notebook_OP.
16 Response/Result/wpURL <to> $b4pInput.Request/inputs/input[1]
17 </to> </copy>
18 </assign>
19 <!-- invocation of HumanTask for method tasks -->
20 <extensionActivity>
21 <peopleActivity name="RefineTheArchitecture"
22 inputVariable="b4pInput" outputVariable="b4pOutput" > ...
23 </peopleActivity>
24 </extensionActivity>
25 <!-- Output assignment to copy the response from HumanTask -->
26 <assign name="RefineTheArchitecture_Output">
27 <copy> <from variable="b4pOutput">
28 <query> <![CDATA[Result[@workproduct="architecture_notebook"]]]>
29 </query> </from>
30 <to variable="RefineTheArchitecture_architecture_notebook_OP">
31 <query> <![CDATA[Result[1]]]> </query> </to> </copy>
32 </assign>
33 </sequence>...
34 </process>

```

Fig. 5.15 Snippet of the resulting BPEL process for the end-to-end example

Using the described transformation, we are able to derive BPEL/BPEL4People process models from MESP method methods. As discussed, these can be deployed into standard BPEL engines and configured to be used with the project team. Figure 5.16 shows the assignment of people (here Alice) to roles defined in the BPEL engine. Figure 5.17 shows a standard repository with an uploaded Requirements Specification that was available at project start.

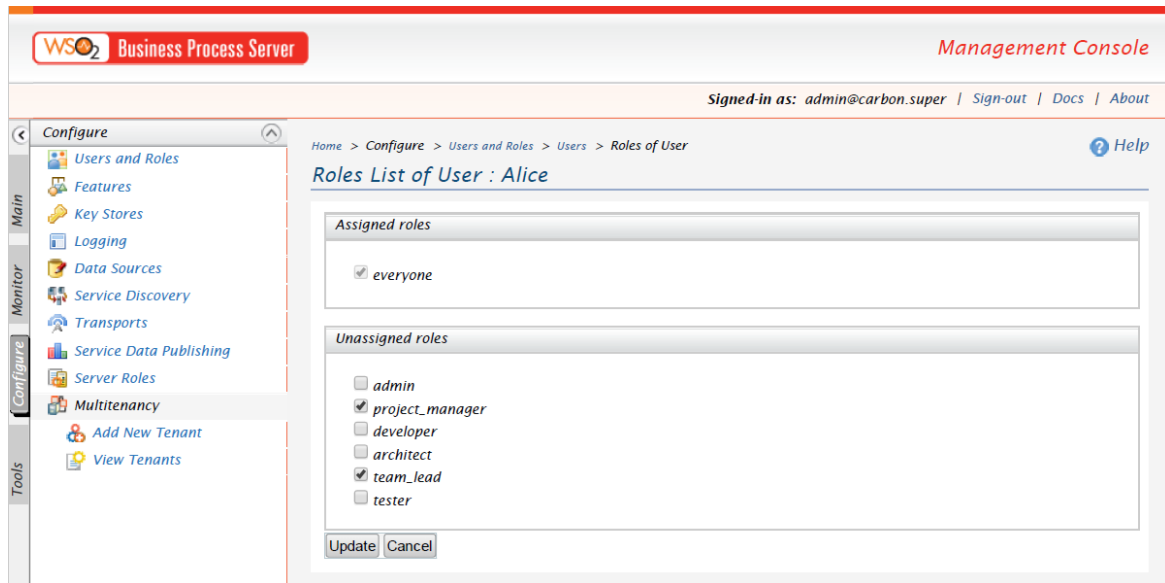


Fig. 5.16 The configuration interface of the BPEL engine to configure role assignment

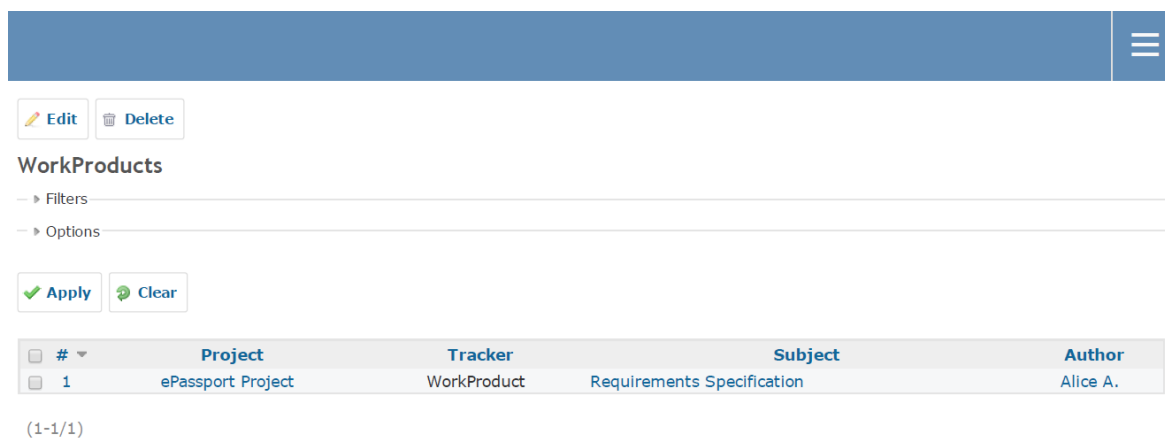


Fig. 5.17 View of the project repository with uploaded WorkProduct

5.6 Summary

In this chapter, we presented the details of method tailoring with our approach that fulfills the MTRs explained in Section 5.1. In particular, we formalized the notion of project characteristics, such that suitable method building blocks can be determined (MTR1). Thereafter, we presented our executable composition language to compose method models. It allows combining method patterns by nesting them into each other (MTR2) and it supports formal control and data flow (MTR3). In order to support the quality assurance of method models, we first defined a consolidated quality model for software engineering method models that categorizes and defines the notion of quality (MTR4). Based on the quality model, we defined an automatic quality analysis that reports inconsistencies, including method pattern violations (MTR5). In the following, we described the automated transformation of method models to process models such that they can be executed with existing standard process engines (MTR6). We explained that as part of the process model, the required HumanTask web services and their GUIs for the project team members are also created (MTR7).

In the following chapter, we discuss the method enactment with our approach and explain requirements and related work, the coordination of activities among the project team, the performance of tasks by single project team members, and the reflection of the method as a foundation for the improvement of method building blocks.

CHAPTER 6

Method Enactment

In the previous chapter, we presented the details of method tailoring by the project method engineer. In this chapter, we discuss the details of method enactment by the project team.

This chapter is structured as follows. We first discuss the requirements and related work in Section 6.1. Thereafter, we discuss the coordination of activities and the support provided by the executed process model (transformed from the composed method model) in Section 6.2. In Section 6.3, we then discuss the performance of individual tasks and how that is supported by the executed process model and the task management component of the BPEL engine. In the following Section 6.4, we discuss the reflection of the method enactment by the project team in order to provide feedback to the senior method engineer. Finally, we conclude the chapter with a summary in Section 6.5.

6.1 Requirements and Related Work

6.1.1 Requirements

6.1.2 Related Work

6.2 Coordinate Activities

6.3 Perform Tasks

6.4 Reflect Method

6.5 Summary

6.1 Requirements and Related Work

In this section, we describe the requirements and related work of method enactment. We first present the method enactment requirements (MERs) that are a refinement of the SRs presented in Section 2.2.1. Then we briefly summarize the related work that will be discussed also in the respective sections later.

6.1.1 Requirements

In this section, we discuss the requirements with respect to method enactment for a holistic solution for software engineering method management based on an assembly-based method engineering approach.

The duty of the project team is to enact the method that is formalized as a method model in order to create the software system. They have to do three things: first, they have to coordinate their activities according to the control flow of the method model. Here, they also have to communicate where to find the work products to be used for each activity. Second, they have to perform the individual activity according to its task description. And third, they have to reflect the enactment of the method in order to provide the senior method engineer with feedback.

As stated with SR3.1, a solution for software engineering method management has to support the project team with execution support for the enacted method model. In order to support the coordination of the project team activities, we therefore derive the MER for our solution that is to *coordinate the activities of the project team according to the method model (MER1)*. The solution has also to *coordinate the data flow between activities (MER2)*.

The SR3.2 describes that the solution has to provide enactment support for the project team via interfaces for the executed method model. In order to support the performance of individual tasks, we refine it into the MER for our solution that it has to *offer task-related GUIs that show information about the current task, the execution state of the method model, and the input work products to be used (MER3)*. Additionally, the *task-related GUIs have to provide means to capture the results of the performed task (MER4)* and the solution has to *offer means to store the task-related results (MER5)*.

As stated with SR3.3 the solution has to provide enactment logs about the enactment of the method by the project team. This shall support the reflection of the method enactment. As the execution of the method model in the process engine reflects the enactment by the team, we refine it into the MER that the solution has to *provide logs for the method model execution and work product alterations (MER6)*.

The discussed MERs are summarized in Table 6.1. Also illustrated is the MESP task where the requirement needs to be addressed. In the following, we discuss each MESP task of our solution for method enactment. We then also explain how the respective requirements are met.

Table 6.1 Method Enactment Requirements and the affected MESP tasks

<i>Reqs.</i>	Description	MESP Task
<i>MER1</i>	coordinate the activities of the project team according to the method model	Coordinate Activities
<i>MER2</i>	coordinate the data flow between activities	Coordinate Activities
<i>MER3</i>	offer task-related GUIs that show information about the current task, the execution state of the method model, and the input work products to be used	Perform Task
<i>MER4</i>	task-related GUIs have to provide means to capture the results of the performed task	Perform Task
<i>MER5</i>	offer means to store the task-related results	Perform Task
<i>MER6</i>	provide logs for the method model execution and work product alterations	Reflect Method

6.1.2 Related Work

Regarding the coordination of activities, there are only few approaches that offer support by executing method models. The tool Demacrone [Har97] and the Guidance Engine of MENTOR [SRG96; Pli96] offer execution support according to the authors. However, both approaches are based on low-level modeling languages and outdated technologies. A more standard-conforming way to allow for execution support is to follow our approach and to extend SPEM in order to overcome its lack of executability. Ellner et al. [Ell+10; Ell+11] implemented their own process engine for their extension eSPEM. Bendraou et al. [Ben+07] sketch mappings to the executable process description language BPEL for their extension xSPEM. For all the described approaches, the extend of the control flow and data flow support is not described. Other approaches like V-Modell XT do not offer execution support, but offer the creation of supporting material, e.g., project plans [KTF11]. In [KKT14], Kuhrmann et al. describe the Process Enactment Tool Framework (PET) that allows transforming a given method model into project templates that project tools can work with. For example, method models based on V-Modell XT can be transformed

into process templates for the Team Foundation Server²² or work product document templates.

Regarding the performance of tasks, except xSPeM and PET, the above approaches provide information to the project team member for the task at hand, e.g., task descriptions. Other approaches do not offer dynamic support, but generate documentation based on the method model. For example, tool like the EPF Composer or IBM Rational Method Composer or MC Sandbox [KÅ11; KÅ12] create static websites that can be browsed for description of processes and tasks (cf. Figure 3.9). A third group of approaches generate specific CASE environments based on the method model, e.g., MERU [PS97], Method Management Tools [KLR96], or MOSKitt4ME [Cer+11]. These CASE environments are specialized to the work products that need to be created as part of the method and thus help the project team member to perform her task.

Regarding the reflection of method enactment for the improvement of method building blocks, there is related work that investigates the enactment of methods. However, much attention is spend on whether the method is enacted correctly, not on whether the method suits to the situation in the first place [Hen+14]. Thus, approaches to measure the degree of formality and optimization of a software development process, so-called *capability maturity models* like SPICE [Loo07] and CMMI [Sof10] are less suitable to address the question of whether the method model or parts of it are appropriate for specific situations. Beside these more heavy-weight, organization-driven approaches, there are more lightweight, team-driven approaches. Postmortem reviews collect experiences from projects that either are completed or have finished a major activity or phase [Dino5]. Here the project team reflects on its experience and documents lessons learned. Based on the same idea, recent agile methods include regular activities to reflect the method enactment and adjust the method for future iterations. For example, Scrum [SS13] includes a regular, informal meeting called retrospective in each iteration. We discuss an approach for the systematic information exchange between the project team and method engineers in the context of software migration methods in [Gri+14].

6.2 Coordinate Activities

In this section, we discuss how our solution addresses the coordination of the activities of the project team according to the method model (MER₁) and the coordination of the data flow between activities (MER₂). In Section 5.5, we already explained how the method model is transformed into a process model and how it is executed in a BPEL engine. In the following, we directly discuss the implications for the project team using an illustrative example.

²²<https://www.visualstudio.com/en-us/products/tfs-overview-vs.aspx>

Figure 6.1 shows the beginning part of a method model that is based on the end-to-end example of Chapter 3. Please note that the data flow is mostly omitted due to readability reasons. We also do not show the method descriptors and constrained scopes (but show their content) as they will not be part of the transformed process model. In the following, we briefly explain the execution order of the shown activities.

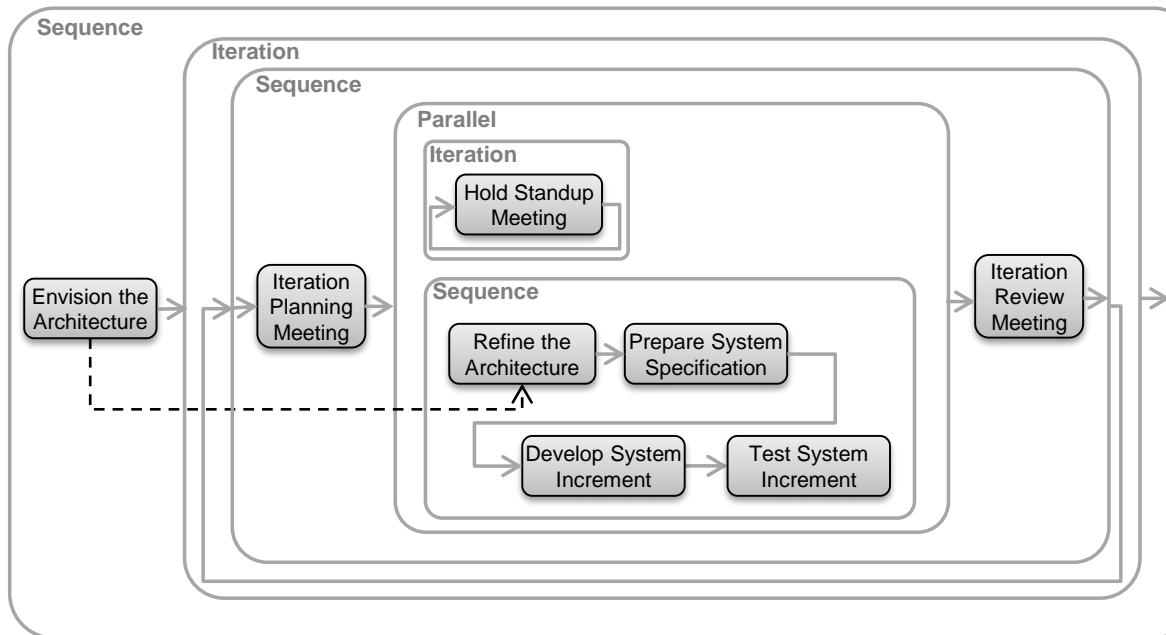


Fig. 6.1 A partial example method model based on the end-to-end-example

The activity *Envision the Architecture* is the first activity that is executed. It is followed by an iteration for the actual development. Within this iteration *Iteration Planning Meeting* is always executed first. Then the activity *Hold Standup Meeting*, shown on the top, is executed repeatedly in parallel to the sequence on the bottom. Within this sequence, the contained activities are executed one after the other. As denoted by the dotted line, the first activity *Refine the Architecture* is receiving its input from *Envision the Architecture*. After the sequence, *Iteration Review Meeting* is executed. If a further run of the iteration follows, it starts from the top with the execution of *Iteration Planning Meeting*. If not, the following activities, not shown in the figure, are executed.

Figure 6.2 shows the process model that is generated from the method model. As depicted, it looks similar to the method model, however some elements are replaced by multiple elements (cf. Section 5.5). For example, each activity that represented a Task, e.g., *Envision the Architecture* was replaced by a triplet of BPEL activities (<assign>, <peopleActivity>, <assign>). Each iteration was replaced by a <repeatUntil> activity with an additional <peopleActivity> contained as the last

element. Not shown is the triplet of BPEL activities that is used to set up the locations of input work products of the process model (cf. Section 5.5) that is executed at the very beginning. Also omitted are some elements that are generated for technical reasons, e.g., namespaces, (cf. [Nee14]).

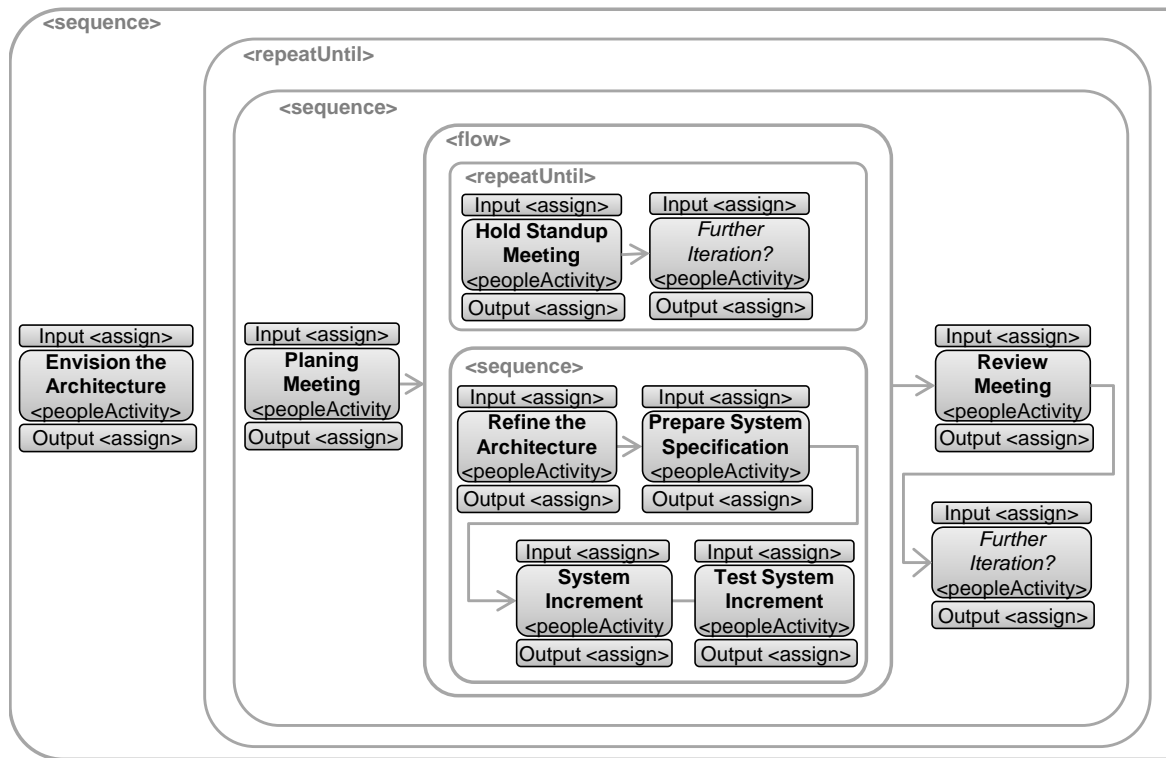


Fig. 6.2 The process model for the method model of Figure 6.1

We illustrate the coordination of activities by the process model execution in the following. Once the outer `<sequence>` is executed, the first triplet of BPEL activities is executed. The activities are executed one after the other. The `<assign>` activity initializes the invocation variable with the Task-related information, e.g., the name “Envision the Architecture” and the associated role “Architect”. The following `peopleActivity` invokes our generic `HumanTask` web service. This creates a workflow task with the provided information in the Task Management component of the BPEL server and the execution stops until a response is received. Team members can log into the BPEL server to see their open workflow tasks according to their roles set up in the system. Thus, now every team member with the role *Architect* can access the created workflow task via the task management view as illustrated by Figure 6.3.

The team member, e.g., Bob, can now claim and perform the workflow task and create the output work product *Architecture Notebook*. He then marks the workflow task as finished and provides the location URI of the output. Once the workflow

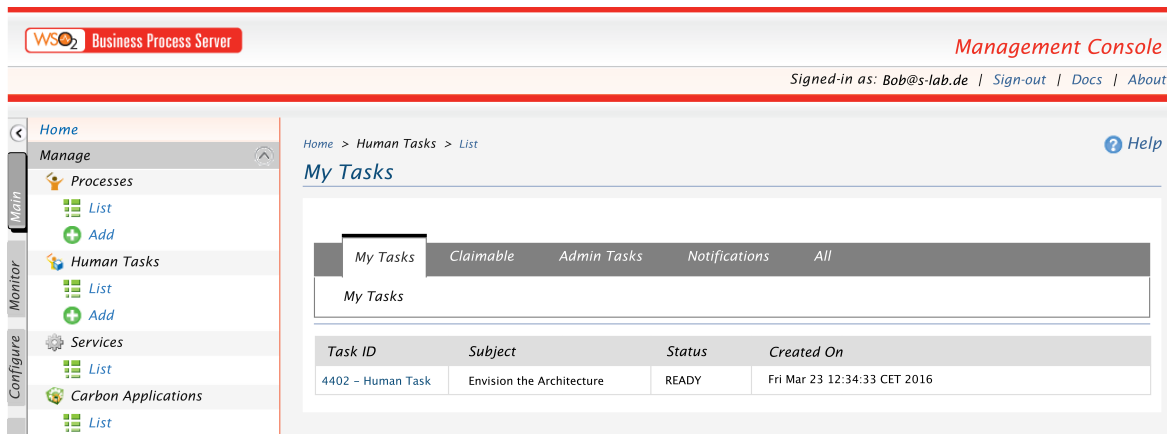


Fig. 6.3 Task management view of architect Bob with a workflow task ready

task is finished by Bob, the execution of the process resumes with the following `<assign>` activity that stores the result into the appropriate result `<variable>` “EnvisionTheArchitecture_architecture_notebook_OP” (cf. 5.15). Afterward, the next activity `<repeatUntil>` is executed. The only child `<sequence>` is executed next and executes its first child, the triplet of activities for the task *Planning Meeting*. Similar to before, a workflow task is created for the associated role *Team Lead* (c.f. 3.10) and the execution is resumed, after it was finished. Next, the `<flow>` activity is executed and it executes its two children in parallel, the `<repeatUntil>` activity and the `<sequence>`. Thus, two workflow tasks are created, one for *Hold Standup Meeting* and one for *Refine the Architecture*. The workflow task that is created for the latter contains the location URI that was provided by Bob after performing the workflow task for *Envision the Architecture*. Based on the specified data flow (cf. Figure 6.1), it is copied from the `<variable>` “EnvisionTheArchitecture_architecture_notebook_OP” during the input `<assign>` activity. Both BPEL activities execute their child activities sequentially, but independent of each other. Once the workflow task for *Test System Increment* is finished, the `<sequence>` is finished. The `<repeatUntil>` activity, however, might be repeated, depending on the outcome of the second `<peopleActivity>`. This creates a workflow task that allows team members with the role *Team Lead* to request a following iteration. A team member, e.g. Alice, can now claim and perform the workflow task that is shown in Figure 6.4. Depending on the decision, the `<repeatUntil>` activity is repeated and a workflow task for *Hold Standup Meeting* is created again.

Only after the `<repeatUntil>` activity and the `<sequence>` are both finished, the triplet of activities for *Review Meeting* is executed and the corresponding workflow task is created. Once that is finished, the `<peopleActivity>` to decide about a further iteration of the development loop is executed.

As described, the enactment of the method is controlled by the BPEL engine via the created and performed workflow tasks. The BPEL engine ensures that the

[<< Back to Task List](#)

Details:

Created On: Mon Mar 26 09:34:32 CET 2016
Updated On: Mon Mar 26 09:41:05 CET 2016
Status: IN_PROGRESS

Description:

Should the Iteration be repeated and another Hold Standup Meeting be performed?

People:

Owner: Alice

Request:

Description Should the Iteration be repeated and another Hold Standup Meeting be performed?
Phase Development Phase
Iteration Development Iteration: 1 HoldStandupMeetings Iteration: 1

Response:

Yes, repeat No

Fig. 6.4 A workflow task to decide about a further run of an iteration

workflow tasks are created in the right order according to the control flow specified in the method model. In addition, it ensures that the right information about input work products is shown based on the data flow specified. Especially in settings, where the coordination of activities is challenging, e.g., in big projects or global software project (cf. [FGS15] and [FSH15]), the project team benefits from this support of the coordination of activities. In general, it is ensured that the enactment of activities does not deviate from the specification in the method model.

6.3 Perform Tasks

In this section, we discuss how our solution provides project team members with information about their task at hand (MER3), how team members store the results of their work (MER4) and how they report them back to the system (MER5). In Section 5.5, we already explained how that is achieved, technically. In the following, we directly discuss the implications for the team member using an illustrative example. In the previous section, we explained how the method enactment is controlled via the creation and performance of workflow tasks by the BPEL engine.

In this section, we want to discuss how task-related workflow tasks are presented to the team members.

First, we focus on the outputs that are to be created. Figure 6.6 shows the workflow task for *Envision the Architecture* as it is presented to the architect Bob after he claimed and started it (see previous Section). As depicted, it shows the information about the associated Task. Based on the provided information, Bob can now create the output work product *Architecture Notebook*. He uploads the result to the project repository as illustrated with Figure 6.5 or specifies it directly there using the provided facilities. Afterward, he provides the location URI of the output, when he marks the workflow task as finished as shown in Figure 6.6.

The screenshot displays a user interface for a project repository. At the top, there is a blue navigation bar with a hamburger menu icon on the right. Below this is a toolbar containing five buttons: 'Edit' (pencil icon), 'Log time' (clock icon), 'Watch' (star icon), 'Copy' (document icon), and 'Delete' (trash icon). The main content area shows a card titled 'WorkProduct #15' for 'ArchitectureNotebook'. The card has a yellow background and includes the following information: 'Added by Bob less than a minute ago.', 'Status: New', 'Priority: Normal', 'Assignee: -', 'Start date: 03/26/2016', 'Due date:', and '% Done: 0%' with a progress bar. Below this, a file entry is shown: 'ePassportProject_ArchitectureNotebook.pdf (2.23 MB) Bob, 03/26/2016 04:54 PM'. At the bottom of the card, there are two sections: 'Subtasks' and 'Related issues', each with an 'Add' button. A second toolbar with the same five buttons is located below the card. In the bottom right corner, there is a text link: 'Also available in: Atom | PDF'.

Fig. 6.5 The Architecture Notebook uploaded to the project repository by Bob

Second, we now discuss the three groups of information provided in a workflow task. Figure 6.7 shows the workflow task for *Refine the Architecture* that is claimed and started by another architect named Eve. As the first group of information, the workflow task shows task-related information that corresponds to the specification given by the senior method engineer (cf. Figures 3.9 and 3.10), e.g., the description of the task and the associated roles. In addition, it shows process-related information based on the process model execution so far. It reflects to which phase and to which run of an iteration the workflow task belongs. For example, the workflow task belongs to the second iteration, while the workflow task shown in Figure 6.6 did not belong to any iteration. As the last group of information, the workflow task shows the location of the input work products so that Eve can access them in the

[<< Back to Task List](#)

Details:

Created On: Fri Mar 23 12:34:33 CET 2016

Updated On: Mon Mar 26 09:01:42 CET 2016

Status: IN_PROGRESS

Description:

To envision a technical approach to the system that supports the project requirements, within the constraints placed on the system and the development team.

People:

Owner: Bob

Request:

Description	<i>This task focuses on envisioning the initial architecture and outlining the architectural decisions that will guide development and testing. It relies on gathering experience gained in similar systems or problem domains to constrain and focus the architecture so that effort is not wasted in re-inventing architecture.</i>
Role	Architect
Steps	<i>Identify architectural goals Identify architecturally significant requirements Identify key abstractions Define approach for partitioning the system Identify interfaces to external systems Verify architectural consistency Capture and communicate architectural decisions</i>
Add. Roles	None
Inputs	Architecture Notebook
Outputs	Architecture Notebook
Phase	Development Phase
Iteration	None

Response:

Architecture Notebook:

<http://redmine.s-lab.de/issues/15>

Complete

Fig. 6.6 The workflow task for Envision the Architecture

project repository. The provided location URI corresponds to the one provided as output for the workflow task *Envision the Architecture* (cf. Figure 6.6). Now Eve can upload a new version of the *Architecture Notebook* in the project repository and provide its location URI.

As described, the information in the workflow tasks support the project team member in performing her individual task. As the information specified by the senior and project method engineers regarding tasks, phases, and iterations are provided right with the workflow task, the project team member does not have to look that information up or ask for it. Same is true for the required input work products of the task whose location is also specified within the workflow task. As with the coordination of activities, this is especially helpful in distributed and large settings, e.g., global software project (cf. [FGS15] and [FSH15]).

6.4 Reflect Method

In this section, we discuss the reflection of the method enactment by the project team and how our solution provide logs for the method model execution and work product alterations (MER6).

In order to support the senior method engineer in improving method services and method patterns, the project team should reflect the enactment of the method and provide her with feedback by capturing lessons learned. Team-based reflection approaches are described for example in [Din05] and [SS13] typically involve the discussion of what went good or bad since the last reflection session.

To support this analysis, it is helpful to have access to information about the actual method enactment. Based on our experience (e.g. [FSH14]), the actual enactment of the method can very often only be reconstructed indirectly and in a tedious manner, based on the created work products and the individual memories of the project team members. If a project repository is used (as designated with our solution), the change log for work products as illustrated by Figure 6.8 is a helpful extension.

However, the performed tasks themselves are typically not documented. To the contrary in our solution, every performed task is associated with a workflow task. This workflow task has to be started and finished by the project team members. Therefore, the log file of the BPEL engine can be used to reconstruct the method enactment as it logs the start and end of workflow tasks. As the raw data log will contain technical details and BPEL terminology, it is less suitable for the direct use by the project team members and should be preprocessed. Yet, such a preprocessing remains future work for our solution.

The results of the method reflection are not (only) to be implemented by the project team itself, but need to be communicated to the senior method engineer. Here, oral feedback and interviews are preferred (cf. 4.2.2) as this is efficient and

[<< Back to Task List](#)

Details:

Created On: Mon Mar 26 14:04:03 CET 2016

Updated On: Mon Mar 26 15:27:01 CET 2016

Status: IN_PROGRESS

Description:

To make and document the architectural decisions necessary to support development.

People:

Owner: Eve

Request:

Name	Refine the Architecture
Description	This task builds upon the outlined architecture and makes concrete and unambiguous architectural decisions to support development. It takes into account any design and implementation work products that have been developed so far. In other words, the architecture evolves as the solution is designed and implemented, and the architecture documentation is updated to reflect any changes made during development.
Role	Architect
Steps	Refine the architectural goals and significant requirements Identify architecturally significant design elements Refine architectural mechanisms Define development architecture and test architecture Validate the architecture Communicate decisions
Add. Roles	None
Inputs	Architecture Notebook : http://redmine.s-lab.de/issues/15
Outputs	Architecture Notebook
Phase	Development Phase
Iteration	Development Iteration: 1

Response:

Architecture Notebook:

Complete

Fig. 6.7 The workflow task for Refine the Architecture



Fig. 6.8 The change log of the project repository

allows for follow up questions. However, especially in distributed settings like global software projects (cf. [FGS15] and [FSH15]) this might not be possible. In [Dino5], the creation of a postmortem report is proposed. In [Gri+14], we discuss the use of feedback forms in the context of software migration methods in an industrial case study. The (senior) method engineer creates these feedback forms to guide the kind of feedback she considers helpful. The project team members then can use them to provide feedback without direct communication.

6.5 Summary

In this chapter, we presented the details of method enactment with our approach that fulfills the MERs explained in Section 6.1.1. In particular, our execution support coordinates the activities of the project team by creating and assigning workflow tasks based on the progress within the process model that is generated from the method model (MER₁). Our solution also provides the appropriate work products for a workflow task based on the data flow of the method model (MER₂). In addition, the workflow tasks show the task-related information specified by the senior method engineer for a task and information about the current phase and iteration (MER₃). The project team member can provide the location of the output work products that she created right in the GUI of a workflow task (MER₄) and the storage of work products is realized using a standard project repository (MER₅).

In the following chapter, we discuss the evaluation of our approach that is based on the prototypical implementation of the described solution for software engineering method management.

CHAPTER 7

Proof of Concept Implementation

We implemented the solution presented in this thesis in a research prototype that covers MESP tasks on all three layers of the software engineering method management hierarchy.

This chapter is structured as follows. In Section 7.1, we discuss the technical implementation of the MESP tool support. In Section 7.2.1, we focus on content realized with the tool support. We discuss a case study from the eID domain created in a joint project with HJP Consulting GmbH to test the capabilities of the MESP tool support. In Section 7.2.2, we discuss the scalability of our tool support and in Section 7.2.2, we discuss the results.

7.1 Tool Implementation

7.1.1 Method Content Definition

7.1.2 Method Tailoring

7.1.3 Method Enactment

7.2 Method Composition

7.2.1 Case Study: Certification Issuance Process

7.2.2 Experiment: Scalability Analysis

7.3 Summary

7.1 Tool Implementation

The implementation of our approach is based on three platforms. First, we developed and extended Eclipse-based tooling for all the MESP tasks related to method content definition and method tailoring. This covers the creation and management of method services and method patterns as well as the creation and quality analysis of method models and their transformation to executable process models. For method enactment, we use, second, the off-the-shelf BPEL engine WSO2 Business Process Server²³ and, third, we use the standard off-the-shelf project repository Redmine²⁴. In the following, we explain the implementation of each of the components presented in Figure 3.4.

7.1.1 Method Content Definition

In this section, we illustrate the implementation of the *Method Building Blocks Editor*, the *Method Repository*, and the *Method Repository Browser*.

Method Building Blocks Editor

The Method Building Blocks Editor is used to define basic method elements, method services, and method patterns. In our implementation, we offer a combination of editors that realize this functionality.

First, our tooling integrates the Eclipse Process Framework (EPF) Composer²⁵ that allows modeling SPEM-based basic elements and SPEM-based method models (cf. Figure 7.1). We developed functionality to import tasks, work products, and roles defined with the EPF Composer into our project repository. Thus, we also allow importing basic elements from existing SPEM repositories²⁶.

Second, basic elements, method services, and method patterns can be defined using a tree-based editor that also functions as the Method Repository Browser. This editor was generated from our meta-model and extended with additional functionality using the Eclipse Modeling Framework (EMF) [Ste+09]. Figure 7.2 shows how the method pattern constraint of a method pattern is created using the context menu of the tree-based editor.

Third, besides using the tree-based editor, the process of method services can be specified using visual editors. These editors are the same ones that can be used to specify the process of method models (cf. Section 7.1.2).

²³<http://wso2.com/products/business-process-server/>

²⁴<http://www.redmine.org/>

²⁵<http://www.eclipse.org/epf/>

²⁶e.g. https://eclipse.org/epf/downloads/praclib/praclib_downloads.php

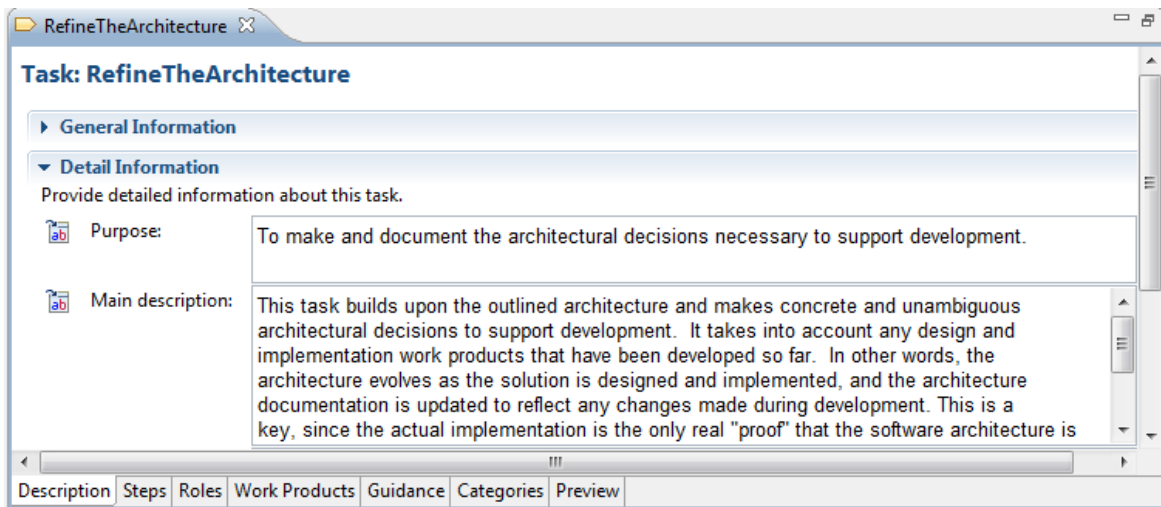


Fig. 7.1 Definition of a Task using the EPF Composer

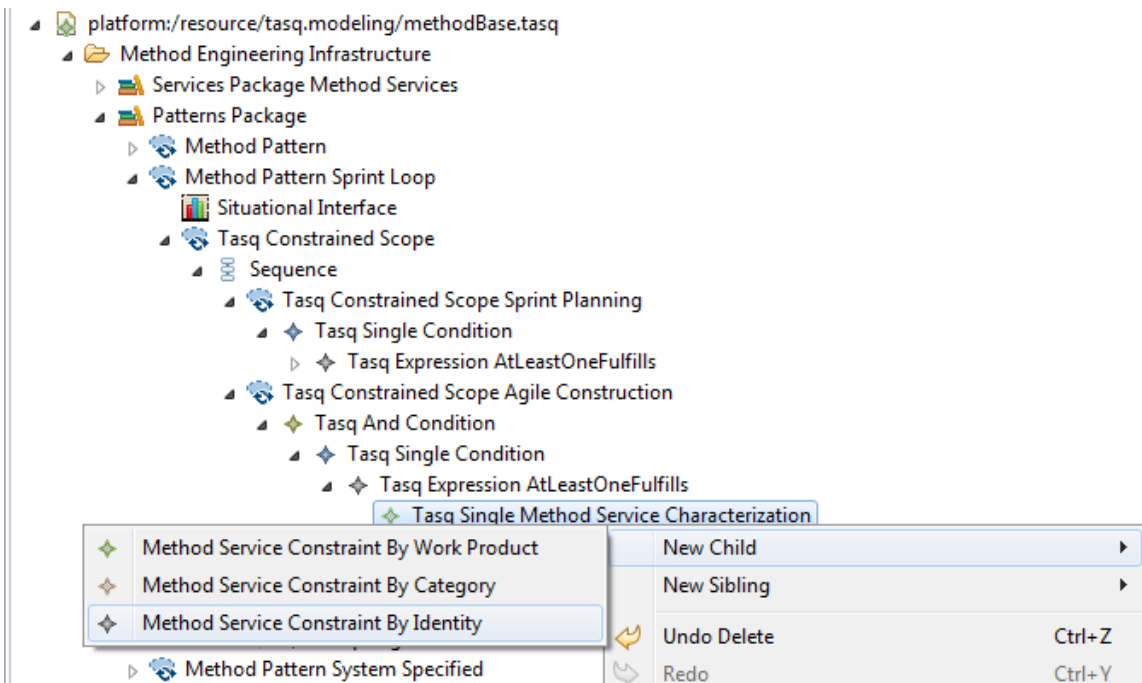


Fig. 7.2 Defining a Method Pattern using the Tree-based EMF Editor

Method Repository and Method Repository Browser

The Method Repository is used to store method content and composed method models. The Method Repository Browser is used to browse through these elements.

The Method Repository is currently a file-based XML storage that is managed by the used Eclipse EMF framework. Multi-user support and performance improvements could be added easily by migrating to the Connected Data Objects (CDO) model repository²⁷ for EMF-based models.

The Method Repository Browser is realized by the tree-based editor presented in the previous section. It allows viewing and editing model elements.

7.1.2 Method Tailoring

In this section, we illustrate the implementation of the *Method Composer*, the *Consistency Checker*, and the *MESP2BPEL Transformer*.

Method Composer

The Method Composer is used to characterize the project and to compose the method model. In our implementation, we offer a combination of editors that realize this functionality.

The tree-based editor discussed in the previous sections can be used to create a new `ProjectMethod` with an empty `Process` and to define the `ProjectGoal` and the `ProjectSituation`. The tree-based editor can also be used to define the contents of the `Process` as shown in Figure 7.3. However, we additionally offer two visual editors to do this.

As the first visual editor, our tooling integrates an extended version of the BPEL Designer²⁸ that originally allows specifying BPEL process models. We extended the editor to support our process-related meta-classes, e.g., task descriptors, method service descriptors, phases, and iterations. Figure 7.4 shows the BPEL Designer with the extended tool palette to add elements to the process. It also shows the extended properties view that shows the Details of the selected `TaskDescriptor`. Figure 7.5 shows a bigger process in the extended BPEL designer. The *Inception Phase* in the figure was collapsed for better overview.

The second visual editor was developed to address the missing capabilities of the extended BPEL Designer to specify and visualize data flow related information. This visual editor is based on the more recent Eclipse Sirius framework²⁹. As illustrated by Figure 7.6, the editor optionally includes data flow related information like required inputs (red rectangles) and existing `InputMappings` (white rectangles)

²⁷<http://projects.eclipse.org/projects/modeling.emf.cdo>

²⁸<http://www.eclipse.org/bpel/>

²⁹<https://eclipse.org/sirius/>

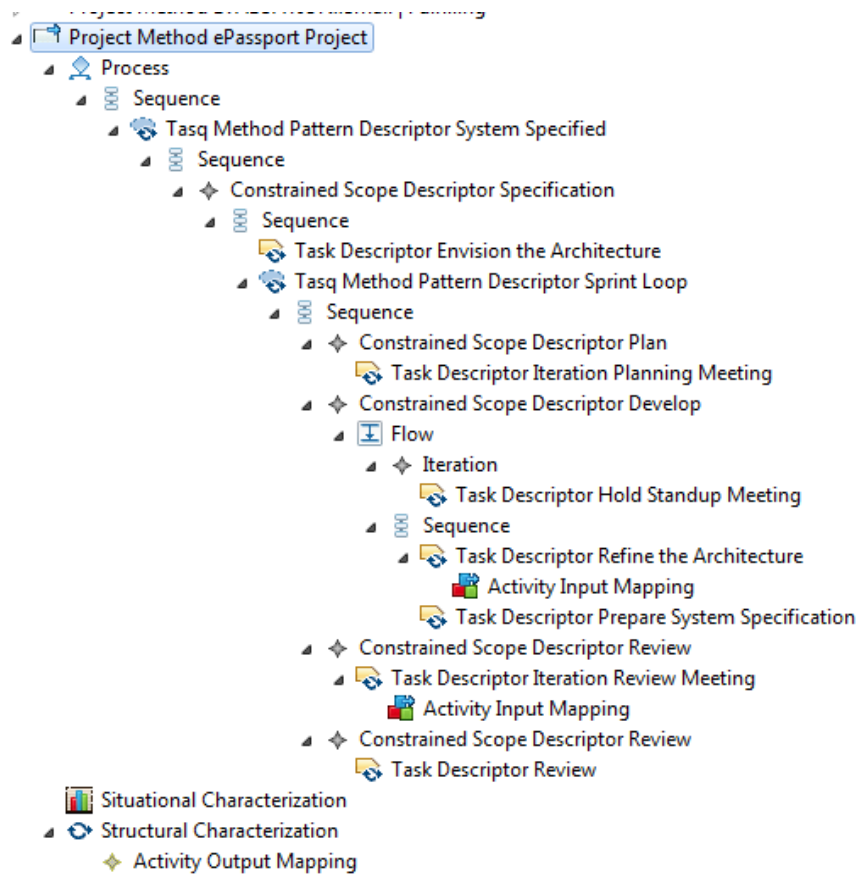


Fig. 7.3 The end-to-end example in the tree-based editor

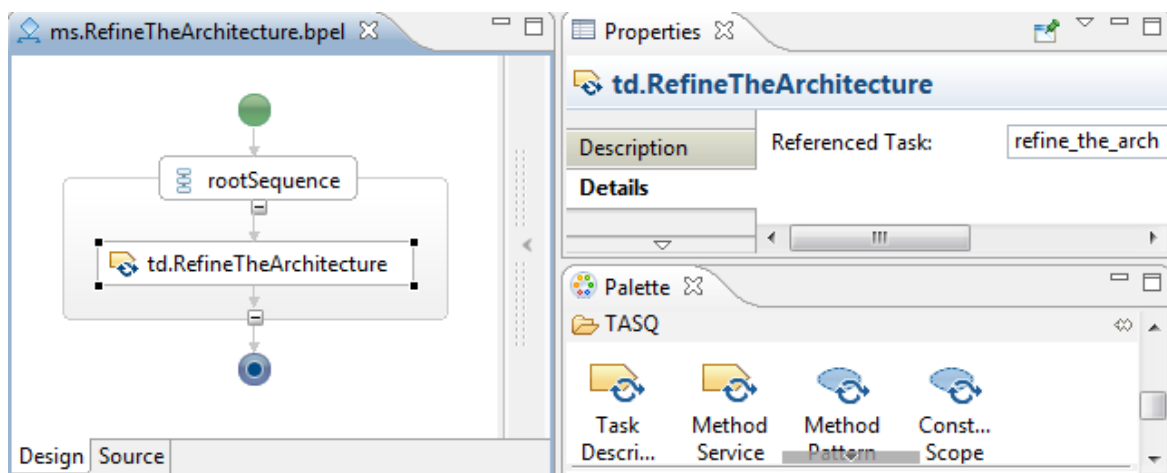


Fig. 7.4 Composition of a process in the extended BPEL Designer

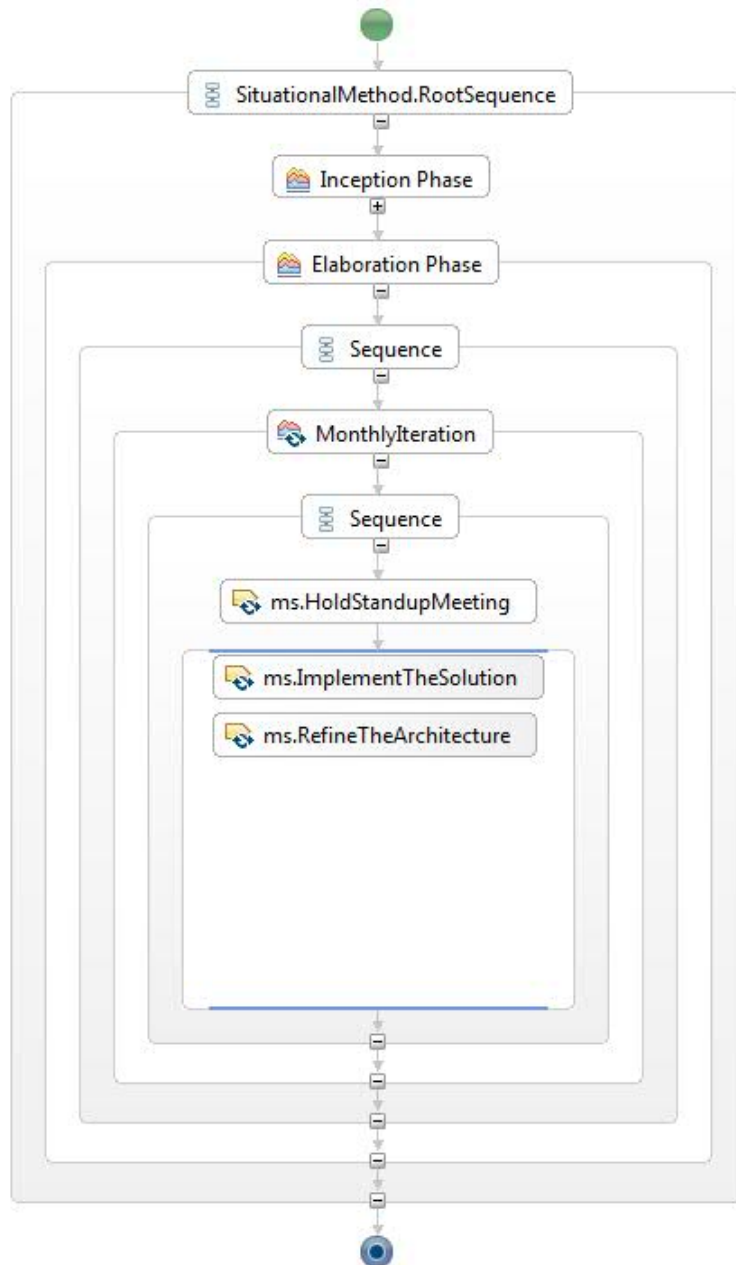


Fig. 7.5 A composed method model in the customized BPEL Designer

or provided outputs (green rectangles). It also features drag-and-drop from the tree-based editor. For example, when a `MethodService` is dragged into the pane, a `MethodServiceDescriptor` for it is created and added.

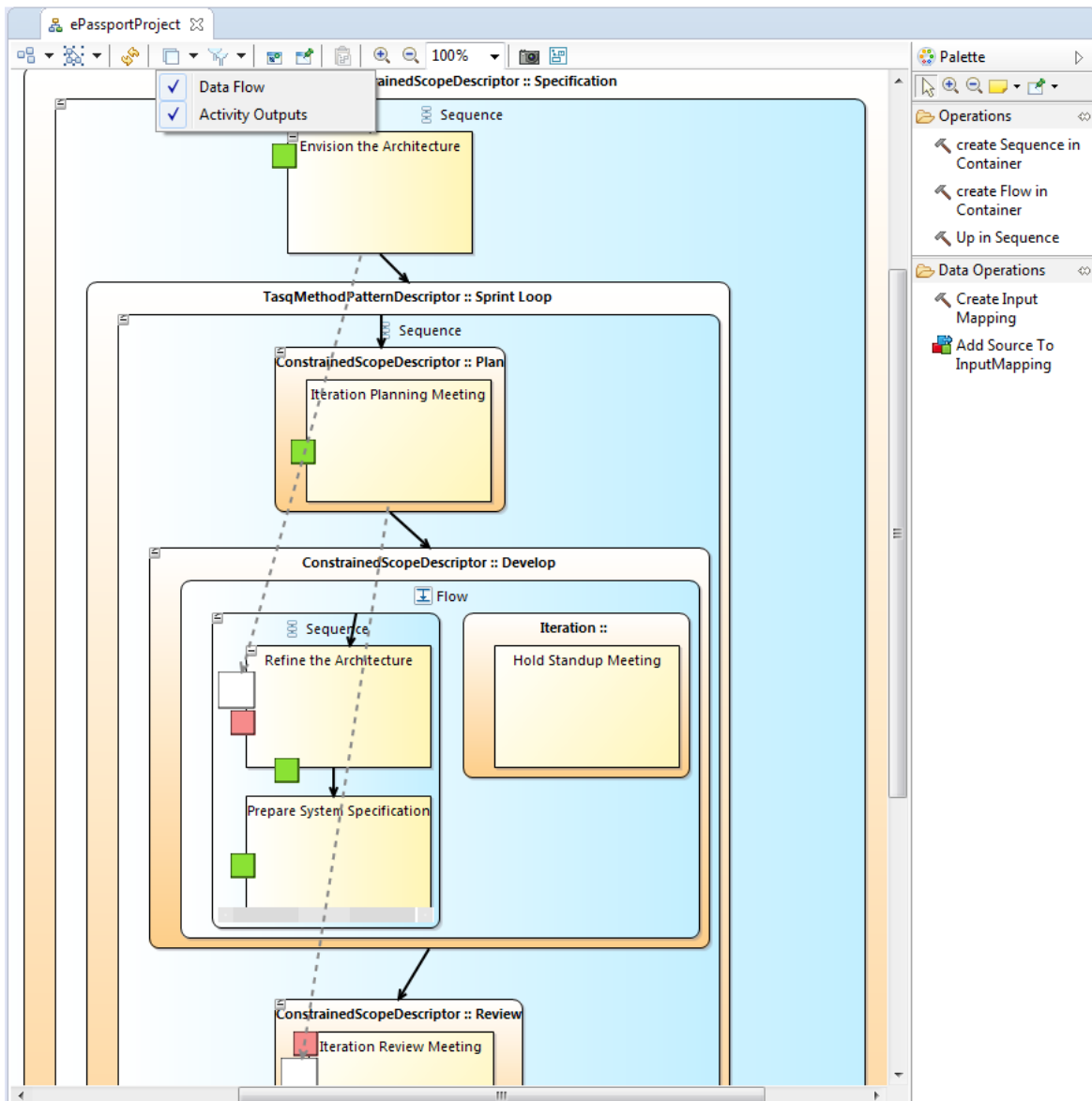



Fig. 7.6 The end-to-end example in the Sirius-based editor

Consistency Checker

The Consistency Checker is used to partially or completely check the composed method model for consistency and quality issues. In our implementation, this functionality is realized by using the EMF Validation Framework that is part of

the Eclipse Modeling Project³⁰. The checks themselves are realized using OCL and Java. The OCL expressions are evaluated with the Eclipse OCL component³¹.

The use of the EMF Validation Framework allows integrating with the existing validation that is provided out of the box and available in all editors. Here, the EMF automatically checks for the conformance of models with their meta-models, so method models are already checked for reference completeness and attribute completeness (see Section 5.4.1).

Figure 7.7 illustrates how detected issues are reported to the user. In the figure, the highlighted Iteration does not contain a HumanDecision that is required to describe the condition for additional iteration runs. Running the validation from the context menu of the visual editor or the tree editor will result in the error reported in the Problem View on the bottom. As shown, the error is visualized in the visual editor with an  mark.

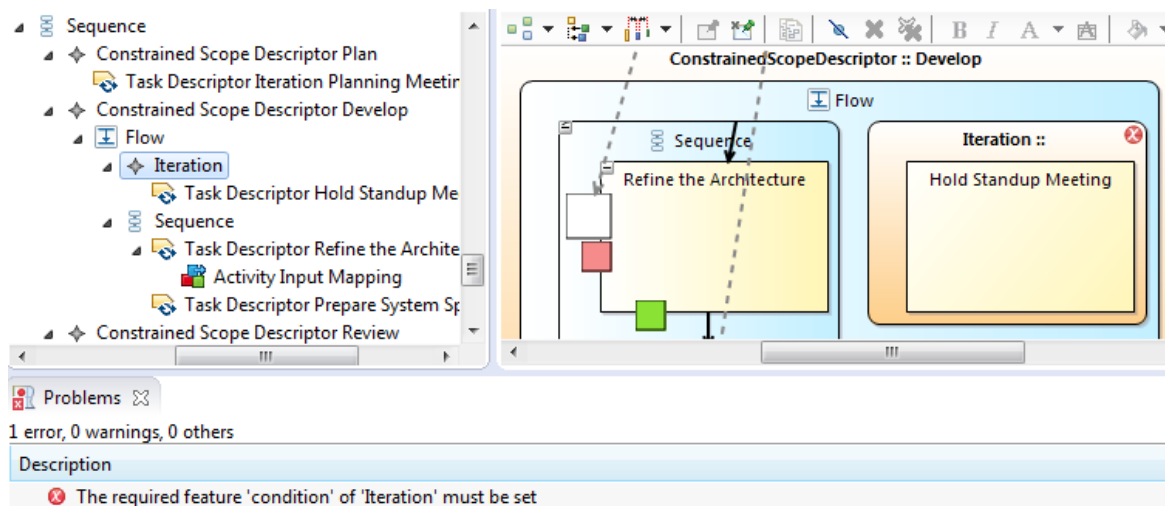


Fig. 7.7 An issue reported in the Problems View and visualized in the Sirius Editor

MESP2BPEL Transformer

The MESP2BPEL Transformer is used to transform the MESP method model into a BPEL/BPEL4People process model. The BPEL/BPEL4People process model consists of two parts, the BPEL process and BPEL4People HumanTask services with their GUIs. In our implementation, the transformation is realized by an Eclipse-Plugin. The functionality can be triggered via the context menu of the tree-based editor upon right click on a `ProjectMethod` as shown in Figure 7.8. The Plugin is written in Java and uses text-based templates. These hold the necessary generic parts of the BPEL/BPEL4People process that are the same for all transformed

³⁰<https://eclipse.org/modeling/>

³¹<http://www.eclipse.org/modeling/mdt/?project=ocl>

method models. They also contain placeholders for the dynamic parts that are generated based on the method model contents. Our implementation deploys the process model automatically into the BPEL server by moving the generated files into specific folders.

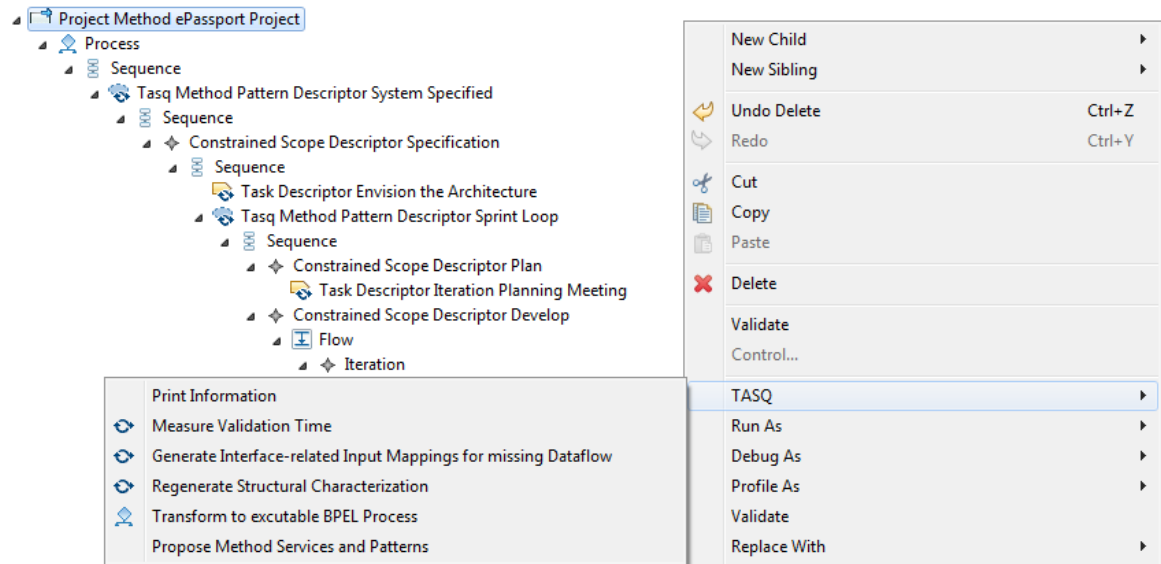


Fig. 7.8 The context menu of the tree-based editor showing the command to transform a method model to an BPEL process model

7.1.3 Method Enactment

In this section, we illustrate the implementation of the *Workflow Engine*, the *Task Engine*, and the *Project Repository*.

Workflow Engine

The Workflow Engine is used to execute the BPEL process and to host the BPEL4People HumanTask services that are invoked by the BPEL process (cf. Section 5.5.1). In our implementation, this functionality is realized by the standard BPEL engine we use, the WSO2 Business Process Server³². Internally, it uses the Apache Orchestration Director Engine (ODE)³³ for the execution of BPEL process models.

Task Engine

The Task Engine is used to manage workflow tasks and serves as the interface to the project team members. In our implementation, this functionality is also

³²<http://wso2.com/products/business-process-server/>

³³<http://ode.apache.org/>

realized by the WSO2 Business Process Server that provides it out-of-the-box. However, the GUIs for the workflow tasks depend on their inputs and outputs and are defined using Java Server Pages³⁴. Therefore, we defined a GUI for each of the three workflow task types (e.g. Figure 6.7). These get deployed to the WSO2 Business Process Server together with their corresponding BPEL4People HumanTask services.

Project Repository

The Project Repository is used to manage work products created by the project team members. In our implementation, we use the standard project repository Redmine (cf. Figure 6.5). In particular, it allows accessing work products via unique URIs. Regarding the third group, a standard project repository is used to store the created work products and create URIs for them.

7.2 Method Composition

In this section, we use the tool implementation presented in the previous section to take more detailed look at the method content definition and method tailoring. The detailed investigation of the method enactment with our tool implementation remains for future work. In the following, we first present a case study that we conducted together with a company. Thereafter, we present a scalability analysis experiment for the performance-critical components of our tool implementation.

7.2.1 Case Study: Certification Issuance Process

As a proof of concept for the MESP tool support, we realized an exploratory case study [RH09] from the eID domain in a joint project with HJP Consulting GmbH. The goal was to demonstrate the approach of MESP and to gain experience with a practical example and to evaluate the capabilities of the tool support.

Case Study Design

The investigation in this case study took place at an eID consultancy and software company called HJP Consulting GmbH. The underlying scenario for the case study is the certification of the eID passport as part of the introduction of a distributed ePassport system. It ensures that the designed eID passports conform to international standards and that they are temper-proof. This scenario was chosen by the participating organization, because it has been very familiar with it. Therefore,

³⁴<http://www.oracle.com/technetwork/java/javaee/jsp/index.html>

it was expected to be easier to identify related concepts like tasks, work products, and roles.

At the time of the case study, the tool support comprised the extended BPEL Designer, but not the Sirius-based editor that was added partly based on the provided feedback. In the following, we describe how the MESP tasks of our approach were performed and illustrate the created model elements. Afterward, we discuss the results of the case study.

Conduction of Case Study

From the case study, the MESP tasks of method content definition and method tailoring were performed by members of HJP Consulting GmbH with feedback and guidance by us.

Method Content Definition In order to *extract reusable method content*, two domain experts of the participating organization captured the existing processes for the certification. They documented their results with the Business Process Model and Notation (BPMN) [OMG11] using the modeling tool Enterprise Architect³⁵. The created BPMN diagram included work products, typical activities, conditional branches and branching conditions. Then, based on feedback by us regarding gaps and inconsistencies, they refined the diagram. Figure 7.9 shows the resulting BPMN diagram and illustrates its size. The process of the diagram basically consists of two parts. First, the required and desired certification is determined and selected. Afterward, based on the selection, the corresponding certification tasks are performed. Thus, the complexity of the depicted process is due to it encoding all possible sequences of activities (i.e. situations).

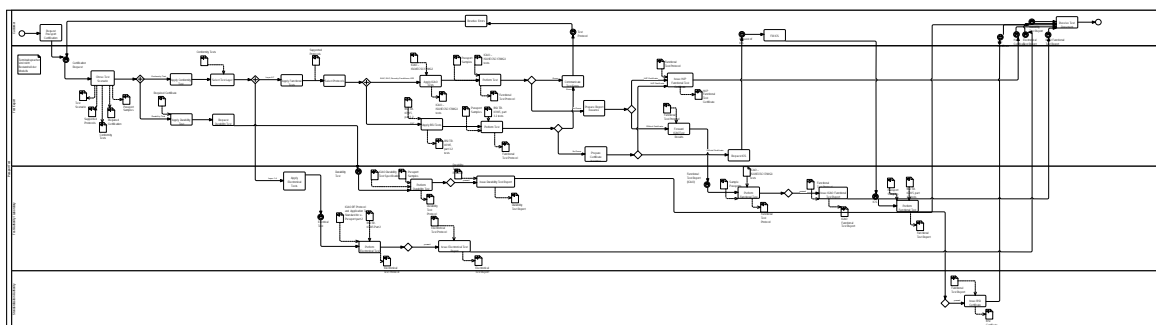


Fig. 7.9 The captured process for the certification of ePassports

As we will see, the complexity of resulting method models is significantly reduced as they are tailored to the situation. In addition, for the case study, the

³⁵<http://www.sparxsystems.com/products/ea/index.html>

organization focused on the modeling of the “happy path”. That is, they chose to ignore alternative flows due to failed certifications for the case study.

Figure 7.10 shows an excerpt of the BPMN diagram. It shows two branches and the icon with the plus sign in the diamond shape indicates that one or both is chosen based on the results of the activity *Select Protocols*. Depending on which path is chosen and for which protocols tests are required, ICAO tests (EAC protocol), BSI tests (BAC protocol), or both (EAC and BAC protocols) are applied and performed. Also shown are input and output work products, e.g., *Passport Samples* are input for *Perform Test*.

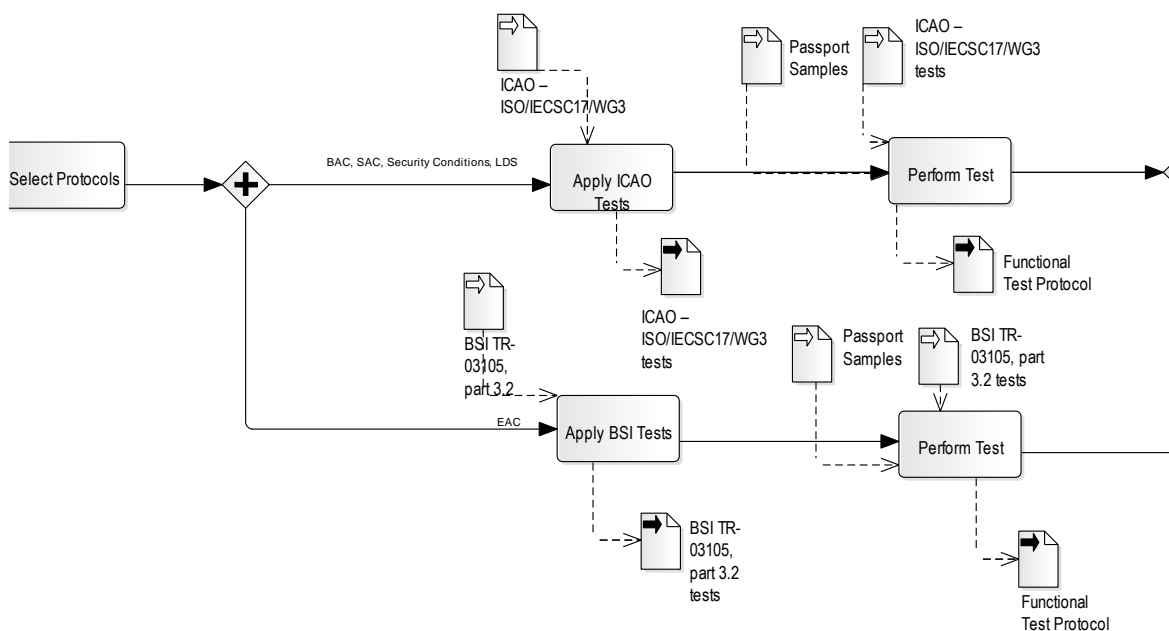


Fig. 7.10 Excerpt of the captured process for the certification of ePassports

In order to *define basic elements*, a third member of the participating organization used the illustrated BPMN diagram. This person was the one that operated the proof of concept implementation. She served both as senior method engineer and defined basic elements and method services. Later she also composed the method model. Based on the information captured in the BPMN diagram, she defined the work products in our tooling as shown in Figure 7.11. In a similar manner, she defined the activities and roles. (Figures 7.11 and 7.12).

Regarding situational factors, no dependencies based on the situation have been known at the time of the case study. The performed activities depended solely on the necessary output work products. Therefore, no situational factors have been defined. Alternatively, the protocols to certify (e.g. EAC or BAC) might have been modeled as situational factor values. The according method services might then have been associated with the corresponding protocol.

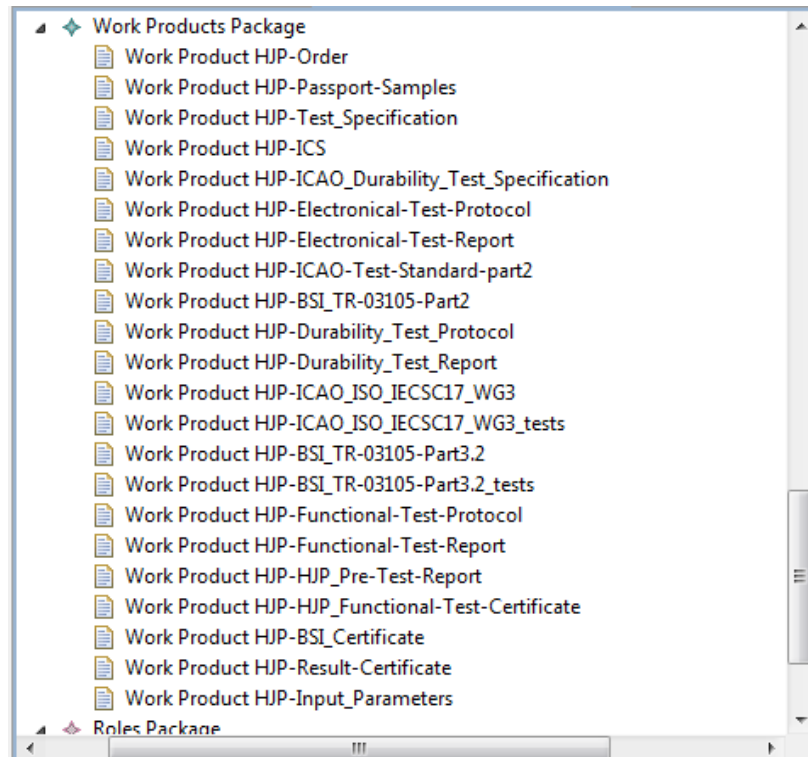


Fig. 7.11 The work products defined based on the BPMN process diagram

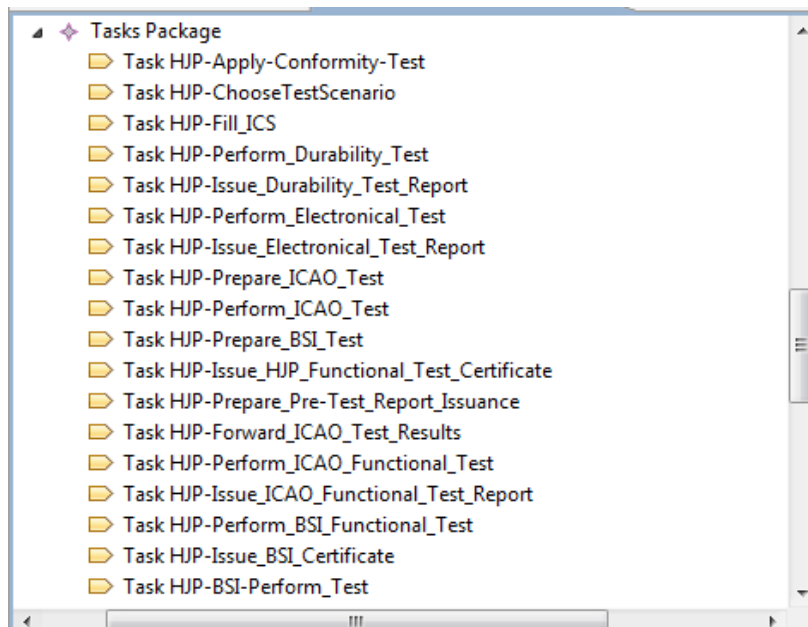


Fig. 7.12 The tasks defined based on the BPMN process diagram

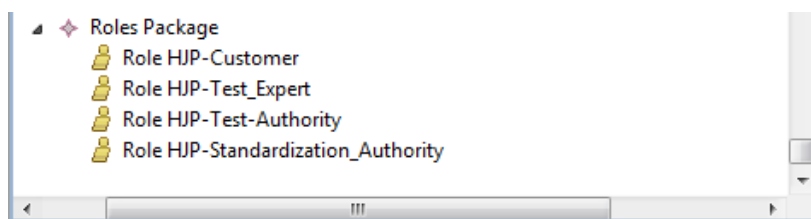


Fig. 7.13 The roles defined based on the BPMN process diagram

The definition of basic elements was seen as the most labor-intensive work in the sense of “lots of clicking, little thinking”. However, in our case study the actual analysis and “thinking” was part of the BPMN diagram creation and thus took place prior to the definition in the tool.

Based on the basic element, the same third member created method services using our tooling and using feedback by us. The resulting method services are shown in Figure 7.14. The defined method services both comprised method services with a single task descriptor and method services containing multiple task descriptors. Thus, there were less method services defined than tasks.

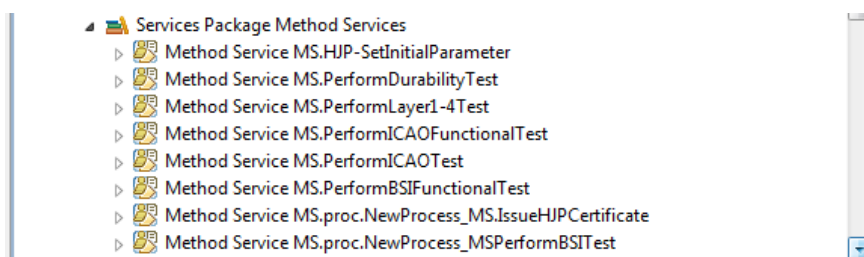


Fig. 7.14 The derived method services for the case study

Figure 7.15 shows the definition of the method service *Perform ICAO Test*. It contains a sequence of three task descriptors, where the last two contain each two input mappings. Also defined are output mappings. The last output mapping for the functional test protocol is highlighted in the Properties View of the tooling and its work product type (Output) and source task descriptor are shown.

Method patterns were not identified in the case study; as the required and provided work products would already predetermine the required method service descriptors for a method model.

The creation of method services and sizing them regarding the contained task descriptors was perceived as being considerable cognitive, but doable work.

Method Tailoring For the creation of the method model, the third member of the organization postulated a typical certification project, where the passport samples had to be certified for the protocol BAC using ICAO tests. She created the method model herself, while we gave feedback and answered clarification questions.

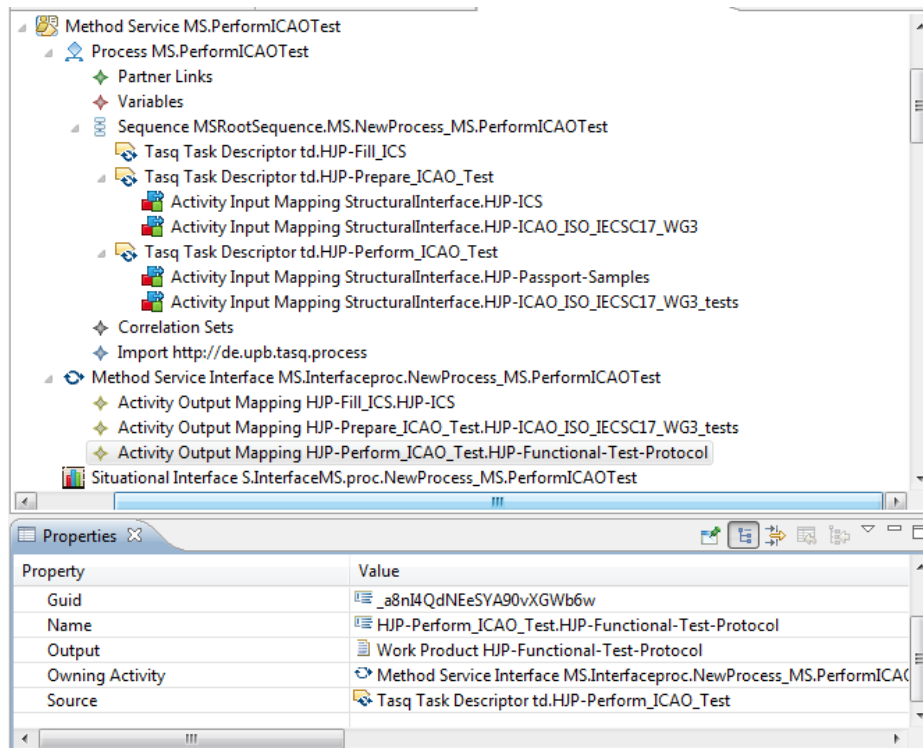


Fig. 7.15 Details of the method service Perform ICAO Test

In order to *characterize the project*, she selected the required output work products of the method model and the provided inputs from the method repository. As explained, situational factors were not defined for the case study.

In order to *compose the project-specific method* and to *assure the quality of the method*, she chose the suitable method services based on the required output work products. The resulting method model is shown in Figure 7.16. Due to the availability of suitable method services with multiple tasks descriptors, the resulting method model only contains two method service descriptors. Also illustrated in the figure are the created input mappings and output mappings for the data flow. Here the automated quality analysis supported her in the identification of missing specifications.

In order to *initialize the method* the third member of the organization invoked the transformation to the BPEL/BPEL4People process model and the deployment of the method model. She also started the process execution and demonstrated the creation of workflow tasks to the other members of the organization.

Overall, the method tailoring was perceived as being doable.

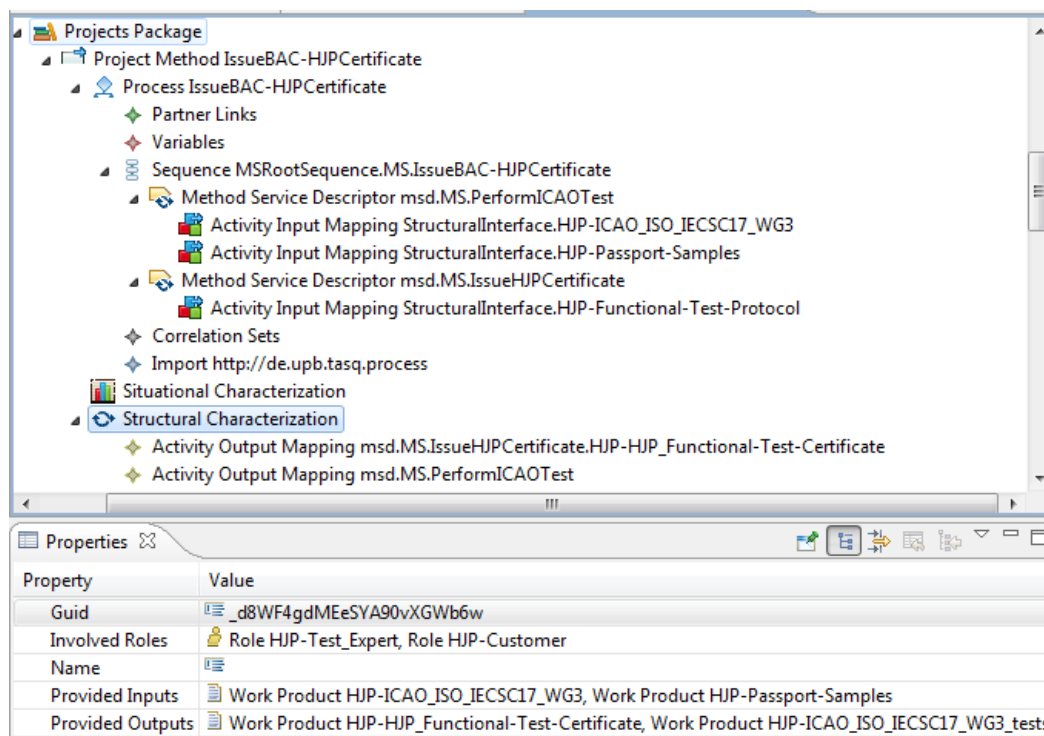


Fig. 7.16 The composed Method for the certification of ePassports

Discussion

In this section, we illustrated a case study from the eID domain from a joint project with HJP Consulting GmbH. The case study was the certification of the eID passport as part of the introduction of a distributed ePassport system. In the case study, three people from the organization were involved, where the tool-related modeling part was performed by the third member of the organization. The following discussion is based on her feedback.

In general, she perceived the creation of method content and the composition of the method model based on the BPMN diagram as easily doable and a good approach for the future. Most of the time for the modeling was spent on the creation of work products, roles, and tasks. While that work was seen as simple, the creation of method services with the right size and content was seen as the most challenging and cognitive intensive work. Together with feedback and clarifications, the modeling took about twelve hours including the time to explain the tooling. The resulting method repository contained four roles, 18 tasks, 22 work products, and 7 method services. The tool support was perceived as being stable and usable. However, it was proposed to extend the visual user interface with the possibility to define and present data flow. This is now possible using our Sirius-based editor.

Overall, the case study shows satisfactory results for our approach and our tool support. Still, as with every case study there are some limitations. In the following, we want to discuss the threads to validity.

The case study was limited to a rather narrow scenario of the certification of ePassports. Here, the creation of basic method elements was perceived as easy. However, it could become more difficult with a bigger number of elements, e.g. to ensure that there are no two work products that describe the same concept. In addition, in the case study, situational factors and method patterns were not defined and used. Therefore, their effect on the creation of method models needs further investigation. Furthermore, our case study was limited to the method content definition and method tailoring, while the method enactment was out of scope. Nevertheless, the case study provides a first indication for the usage of our approach and the provided tooling. The case study was not performed by us, but by IT consultants of another organization and based on a real-life scenario.

7.2.2 Experiment: Scalability Analysis

Beside the investigation of the approach and the tool support in a case study, we also analyzed the scalability of our tool support. Our goal was to determine whether it scales well enough, even for very big method models.

Experiment Design

In our scalability analysis experiment, we focused on the two performance-intensive algorithms that are part of our tool support. The first algorithm is the automated quality analysis with the Consistency Checker that is also highlighted in [FK16]. The second algorithm is the transformation of method models to BPEL/BPEL4People with the MESP4BPEL Transformer. We investigated their performance on a regular workstation with respect to method models with varying properties.

Process Model Data Sets For the scalability analysis, we composed method models from a method repository that had been mainly derived from the practices library of the EPF³⁶. Based on prior experiments, we prepared method models with varying parameters focusing on the parameters that influence the runtime of the algorithms. The computational complexity for the MESP2BPEL Transformer is linear in the number of method model elements including the (transitively) referenced elements in the method repository. Therefore, it would have been sufficient to create method models with a varying number of method service descriptors. However, the computational complexity of the Consistency Checker depends on various parameters. Especially, for the evaluation of method pattern

³⁶<http://epf.eclipse.org/wikis/epfpractices/index.htm>

constraints, our tooling uses an existing black box component, whose runtime characteristics had not been known to us. As a result, we created the three data sets A, B, and C of models as shown in Table 7.1. The test method models are designed such that they represent the worst case for the analysis of the Consistency Checker. For example, if a certain method service descriptor is required, it is not included in the model, such that the algorithm cannot terminate before analyzing the whole model.

Table 7.1 The data sets of process models for the scalability analysis

<i>ID</i>	<i>Service Descriptors</i>	<i>Input Mappings</i>	<i>Constrained Scopes</i>	<i>Partial Constraints</i>	<i>Descriptors per Scope</i>
<i>A1</i>	500	-	-	-	-
<i>A2</i>	500	1000	-	-	-
<i>A3</i>	500	2000	-	-	-
<i>B1</i>	500	2000	5	5	100
<i>B2</i>	500	2000	5	15	100
<i>B3</i>	500	2000	20	60	25
<i>C1</i>	2000	4000	-	-	-
<i>C2</i>	2000	4000	20	60	100

We created a *set A* of method models that do not contain method pattern descriptors and thus no constraints that need to be checked as part of the quality analysis of the Consistency Checker. The method models of set A vary in the number of input mappings that they contain. In addition, we created a *set B* of method models that contain a varying number of method patterns. While the number of method service descriptors and input mappings is constant, the number of constrained scopes, partial constraints, and method service descriptors per constrained scope vary. Both sets exceed the complexity of method models based on software engineering method frameworks like RUP [Kru99] or V-Modell XT [HHo8] and represent an upper bound for realistically sized method models. The *set C* of method models that we created contains even bigger method models that are unrealistically big and aims primarily at investigating the boundaries of the scalability of the Consistency Checker and the MESP2BPEL Transformer.

Execution Environment The scalability analysis was carried out on a workstation with an Intel Xeon W3530 CPU and 16 GB of RAM. It ran Windows 8 and the Eclipse IDE Version Juno.

In the following, we first describe how we conducted the scalability analysis. Afterward, we conclude with a discussion of the scalability analysis experiment.

Conduction of Experiment and Results

With the data set of method models, we evaluated the scalability of the Consistency Checker and the MESP2BPEL Transformer. For each method model, we took the average of 10 runs, although the measurements deviated only in fractions of a second. In the following, we present and discuss the results.

Analysis of Consistency Checker We measured the time between the invocation of the automated quality analysis and the output of the results (cf. Section 5.4.2).

Table 7.2 Results of the scalability analysis of the Consistency Checker

<i>ID</i>	<i>Runtime (sec)</i>
<i>A1</i>	0.1
<i>A2</i>	2.0
<i>A3</i>	3.4
<i>B1</i>	3.8
<i>B2</i>	3.9
<i>B3</i>	4.0
<i>C1</i>	31
<i>C2</i>	34

Table 7.2 shows the positive results of our scalability analysis. The evaluation shows, that realistically sized models (sets A and B) are analyzed in a few seconds. The method models of set A contain a varying number of method service descriptors and input mappings, but no method patterns descriptors. The number of input mappings directly influences the runtime as shown by the differences between A1 and A2, and also, A2 and A3. The number of method service descriptors also plays a role, as shown by the high increase from A3 to C1.

The method models of set B contain method pattern descriptors with different characteristics. Comparing the results for model A3 to the models B1, B2, and B3, and in addition, the results for the model C1 to the model C2, shows that the analysis of method pattern constraints has a minor influence on the runtime compared to the evaluation of quality characteristics in general. The analysis takes about 3 to 4 seconds for the realistically sized models of set B.

The analysis of the unrealistically big method models of set C, which are about 4 times bigger than the ones of sets A and B, takes about 30 seconds (cf. C1/C2). Here, the analysis for the unrealistically big model C2 takes about three seconds longer than the equivalent model without method pattern descriptors.

The scalability analysis shows that the requirement MTR5.5 is fulfilled and that the performance of the component allows its continuous use during method

composition. For the evaluation, we considered the runtime of the analysis of complete method models. However, the user has the choice to run the analysis on parts of the model to get specific results that are computed quicker.

Regarding threats to validity, there is the risk that the method models that we created are not representative and that the analysis of other method models takes much longer. However, we do not consider this very likely, because of the variety of characteristics we investigated in the described evaluation and in the experiments before. In fact, for the evaluation of pattern constraints, we expect even lower runtimes in practice, because, e.g., the evaluation for an “exists” quantifier stops as soon as one element is found. However, in our evaluation models, we ensured that the evaluation does not stop early, with our choice of pattern constraints and method service descriptors.

Analysis of MESP2BPEL Transformer We measured the time between the invocation of the transformation and the creation of the output files (cf. Section 5.5.1). Running the transformation, even on the unrealistic big models of set C runs in less than a second (cf. Table 7.3).

The computational complexity of the MESP2BPEL transformer is in linear time in the number of method model elements. The runtime depends on the number of elements that need to be considered during the transformation, especially the transformed tasks and input mappings. The MESP2BPEL Transformer scales far beyond the required size of method models.

Table 7.3 Results of the scalability analysis of the MESP2BPEL Transformer

<i>ID</i>	<i>Runtime (sec)</i>
<i>A1 – A3</i>	<1
<i>B1 – B3</i>	<1
<i>C1 & C2</i>	<1

Regarding threats to validity, there is the potential risk that other factors that are not reflected by the test models play an important role for the scalability. However, while parts of the Consistency Checker are a black box for us, the internals of the MESP2BPEL Transformer are known to us. Thus, this is not very likely.

Discussion

In the previous sections, we presented the scalability analysis experiment of our proof of concept implementation. For the analysis, we created method models of different sizes including unrealistically big method models. Our experiment

showed that the Consistency Checker processes these models in reasonable time and that its performance allows its continuous use during method composition. The MESP₂BPEL Transformer processes these method models in under a second. As the transformation is invoked only once at the end of the method composition, this is more than sufficient. Thus, both critical components scale beyond realistically sized models.

Despite the positive results, there are some threats to validity that need to be considered. As mentioned, there is the potential risk that our method models do not represent the variety of method models well. In particular, they might not cover valid worst cases, however, we do not think that this is likely. In addition, for the scalability analysis, we focused on the eclipse-based parts of our tool support and did not analyze the standard components. In particular, we did not analyze the scalability of the used BPEL server. While it would always be possible to switch to a different BPEL server, its evaluation could render a more complete picture.

7.3 Summary

In this chapter, we presented our proof of concept implementation of our tool support that covers MESP tasks on all three layers of the software engineering method management hierarchy. We explained that our tool implementation is based partly on eclipse-based components that we developed and extended and partly on off-the-shelf standard products. With our research prototype, we demonstrated the feasibility of our MESP approach and the validity of the underlying meta-model and related concepts. In addition, we discussed a case study that we conducted together with a company. Here, we got third party feedback on the usage of our tooling, however, in a limited setting. This led for example to improvements for the Method Composer. As the third piece of our proof of concept implementation, we performed a scalability analysis experiment. We explained that it indicated that the tool support scales beyond realistically sized models.

While our proof of concept implementation allows drawing first conclusions, it does not replace a thorough evaluation using our approach in real-world software projects. Performing such an evaluation, however, is beyond the scope and time frame of this thesis as it requires a significant number of these projects in order to deliver reliable results. We therefore propose to consider the evaluation and refinement of our approach within industrial projects as a possible follow-up thesis.

In the following chapter, we summarize the contributions of this thesis, conclude the thesis and discuss potential future work.

CHAPTER 8

Conclusions and Outlook

In this thesis, we have presented our solution for software engineering method management that covers method content definition, method tailoring, and method enactment. Our solution is accompanied by tool support that spans these three levels and that we implemented in a research prototype.

In this concluding chapter, we first summarize our contributions in Section 8.1. Thereafter, we discuss the fulfillment of the stated requirements for a solution for software engineering method management in Section 8.2. Then, in Section 8.3, we conclude with future works for software engineering method management.

8.1 Contribution Summary

The goal of this thesis was to develop a solution for software engineering method management that addresses the entire lifecycle of software engineering method models. Such a solution needs to span all three layers of software engineering method management: First, it has to provide means to define method building blocks for activities and repeatable control flow patterns that reflect reusable method knowledge, like new trends, best practices, and lessons learned. Second, it has to provide means to compose suitable method building blocks to executable method models customized to specific situations. Third, it has to support the proper enactment of these method models.

To achieve the goals of this thesis, we presented our solution for software engineering method management MESP. Figure 8.1 gives an overview of our contributions on each level of the software engineering method management hierarchy. In the following, we shortly summarize these contributions.

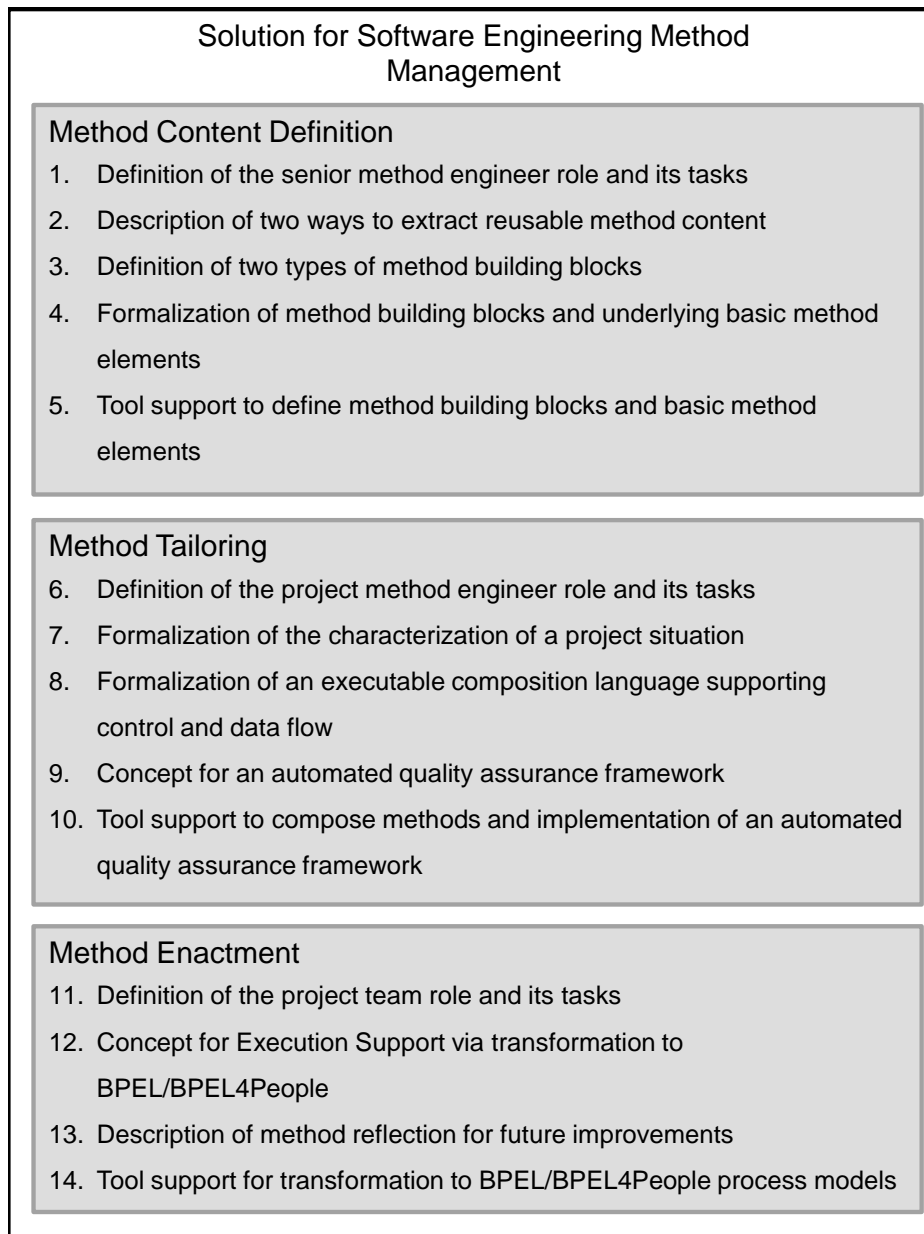


Fig. 8.1 Contribution Overview of our thesis

Method Content Definition

To systematize the method content definition, we defined the role of a senior method engineer with her tasks to extract method content and to define reusable method building blocks.

In order to extract method content, we described two ways to extract reusable method content, one based on the body of literature and one based on the daily practice of organizations.

In order to define method content, we defined two types of method building blocks. First, method services represent atomic and composite activities similar to building block types previously proposed in related literature, however, with a formal interface and executable. Second, the novel notion of method patterns represents abstract control flow patterns that prescribe the use of method services fulfilling specific constraints. Together with the underlying basic method elements that are based on the de-facto standard SPEM, we formalized both types of method building blocks in terms of a meta-model. In particular, we introduced a domain-specific language to express method pattern constraints as part of the meta-model. In addition, we formalized situational factors in order to characterize method building blocks with respect to their suitability for different situations.

For the evaluation of our method content definition, we implemented tool support in a research prototype that allows modeling basic method elements, method building blocks, and situational factors.

Method Tailoring

To systematize the method tailoring, we defined the role of a project method engineer with her tasks to compose a project-specific method model using previously defined method building blocks.

In order to characterize and formalize the project situation, we use the same situational factors that are used to characterize method services and method patterns easing the identification of suitable method building blocks.

In order to compose an executable method model, we defined a composition language that is inspired by SPEM and BPEL. From SPEM, it borrows a similar terminology and the separation between method content and process that fits well to our purpose. From BPEL, it borrows the control flow structure and the semantics of several control flow elements. In particular, our approach also supports (executable) data flow, which is neglected in many other related approaches.

In order to support the project method engineer during method composition, we created a quality analysis framework. In a first step, we defined a quality model for software engineering methods to structure the notion of quality. Then we operationalized different quality characteristics with the quality analysis framework. In particular, we defined an on-the-fly translation of method pattern constraints to

expressions in OCL in order to check for method pattern violation for the method patterns used in the method model.

To prepare the method model for execution, we created a transformation to a standard BPEL/BPEL4People process model including generic HumanTask web services and their GUIs. These GUIs allow project team members to interface with the process model later during method enactment.

For the evaluation of our method tailoring, again we implemented tool support in a research prototype that allows creating a method model, checking it for consistency issues, and transforming it. With the tool support, we conducted a targeted case study that demonstrated the basic applicability of our approach. We also conducted a scalability analysis that showed that the automated quality assurance and the transformation to BPEL/BPEL4People performs sufficiently fast.

Method Enactment

To systematize the method enactment, we defined the role of the project team and its tasks to enact the method and provide feedback to the senior method engineer.

In order to enact the method model, we defined the concept of execution support by transforming the method model to a BPEL/BPEL4People process model. The process model can be executed with a standard BPEL server and the project team interacts with its task management component that manages the workflow tasks for the team that are created during method model execution. Our concept also includes the usage of a standard project repository to store created work products under unique URIs. These URIs are used to represent input and output work products when performing workflow tasks.

In order to support the senior method engineer in refining method building blocks, we described how the enactment logs of the BPEL server and the standard project repository can be used to provide feedback.

For the evaluation of our method enactment, we integrated tool support in a research prototype that in particular allows deploying and executing our transformed BPEL/BPEL4People process models. Besides executing the process model according to the control flow, it requests and shows work product locations in accordance with the method model data flow specification.

8.2 Fulfillment of Requirements

In Chapter 2.2, we described common challenges to software engineering method management. In order to develop a solution that addresses these challenges, we defined solution requirements for such a solution in Section 2.2.1. In this section, we describe how our MESP approach fulfills these solution requirements.

Solution Requirement 1.1 – Abstract Orderings Our approach introduces the notion of method patterns that allow modeling abstract orderings without referencing concrete methods services. Instead, suitable method services are characterized by method pattern constraints, such that method patterns can be used with method services that are created later in time.

Solution Requirement 1.2 – Explicit Interfaces Method services, method patterns, and method models have interfaces that characterize them and abstract from their contents. Based on the formalization of situations with situational factor values and the description of processed work products and involved roles, suitable method building blocks can be identified and composed.

Solution Requirement 1.3 – Composite Building Blocks Method services can describe single, atomic tasks. However, they can also describe composite tasks by encapsulating a process with multiple tasks descriptors or method service descriptors. As these composite method services have the same structure and interface like atomic method services, they can be handled in the same way. Especially, their interface allows abstracting from the possibly complex content.

Solution Requirement 1.4 – Method Repository The method repository in our approach stores all information to describe method models: basic elements, method services, method patterns, and method models. Enactment-related information is stored in the BPEL engine and the project repository. The BPEL execution logs are stored within the BPEL engine and logs about changes to the work products are stored within the project repository.

Solution Requirement 1.5 – High-level Modeling Our approach offers a DSL based on a meta-model and appropriate tooling to model method building blocks. Thus, senior method engineers can focus on the modeling aspect and do not have to deal with low-level languages.

Solution Requirement 2.1 – Project Characterization In our approach, the project context is characterized by a project goal that describes the available and desired work products and a project situation that describes the related situational factor values. Thus, our approach supports the formal characterization of projects.

Solution Requirement 2.2 – Composition Guidance In our approach, method patterns are used as a starting point for the composition of method models. With their constraints, they guide the project method engineer in composing the method. The formal project characterization and the explicit interfaces of method services

and method patterns also guide her in the choice of method building blocks. In addition, our automated quality analysis points her to inconsistencies that need to be resolved, especially violated method pattern constraints.

Solution Requirement 2.4 – High-level Composition Our approach offers a DSL based on a meta-model and appropriate tooling to compose method models. Thus, project method engineers can focus on the modeling aspect and do not have to deal with low-level languages. In particular, they can use visual editors to compose method models.

Solution Requirement 3.1 – Executable Process Method models created with our approach can be executed with a BPEL server. For this purpose, method models are automatically transformed to equivalent BPEL/BPEL4People process models.

Solution Requirement 3.2 – Enactment support with Human Interface Our enacted BPEL/BPEL4People process models create workflow tasks that are accessed by team members via the task management component of the BPEL server. It shows task- and enactment-related information and allows the team member to specify and access the location of work products related to her task.

Solution Requirement 3.3 – Enactment Logs As out-of-the-box features of the BPEL engine and the project repository execution and change logs captures enactment-related information.

As described in this section, our solution for software engineering method management fulfills all stated solution requirements. Thus, it addresses the described challenges of method content definition, method tailoring, and method enactment.

The challenge to enable updates of method content based on new trends, best practices, and lessons learned is addressed by well-defined, interconnected basic elements and method building blocks that reflect different levels of granularity. The challenge to enable the creation of consistent software engineering methods for specific situations based on defined method content is addressed by the situation-specific composition and quality assurance of method building blocks. The challenge to enable the proper enactment of the tailored method according to its definition is addressed by supporting the proper enactment of the method model with a BPEL process engine and automated transformation to BPEL/BPEL4People models.

8.3 Outlook on Future Work

In this thesis, we presented a holistic approach for software engineering method management. However, we identified room for further extensions and improvements as well as follow-up research. In this section, we want to discuss possible extensions of our approach and present ideas for follow-up research.

Modeling Language Extensions With our approach we provide an end-to-end solution for method content definition, method tailoring, and method enactment. Possible extensions are based on the current capabilities of our modeling language that is defined by our meta-model. In the following, we discuss two language extensions.

One possible extension is the introduction of language concepts for exceptional control flows. While the specification of software engineering methods traditionally focuses on the “happy path”, the specification of exceptional flows is quite common for business processes and for executable process description languages like BPEL. In addition, the project management domain knows concepts like escalation to resolve process-related issues. As our approach allows executing method models, it would benefit from the definition and execution of exceptional flows. This would mean to specify the control flow, e.g., when activities exceed their designated time frame (time box).

Another possible extension is the introduction of artifact lifecycles. In our approach, we focus on activity-based modeling of methods that is the predominant approach. However, especially to ensure that activities are performed properly, some methods evade to artifact-centric modeling, e.g. V-Modell XT. Here, the progress is determined by the individual lifecycle states of related artifacts, as described for example in [FGS15]. Artifact lifecycles could be to refine interface specifications of tasks and method services. They could also extend method pattern conditions and allow conditional control flow based on artifact states during runtime.

Quality Analysis Extensions The current implementation of our quality analysis uses a static analysis approach. Currently, conditionally executed parts of the method model are treated like regular parts. The expressiveness of our quality analysis could be improved by taking into account the conditionality. For example, the quality analysis should warn the project method engineer, if all sources of a required input work product are only conditionally executed (within a different execution branch than the requiring target). This might require the integration of model checking techniques. Additional modeling language extensions would also require extensions to the quality analysis, for example to detect contradiction between the specified artifact lifecycles and the used activities (c.f. [Wah+09]). Lastly,

as part of this thesis, we did not formalize all proposed quality characteristics, as this is still an open research question especially for situation-dependent and non-critical quality characteristics.

Tool Support Extensions The application of a holistic solution for software engineering method management is not feasible without proper tool support. As part of this thesis, we developed a research prototype to demonstrate the feasibility of our approach. For productive use in industrial projects, it needs especially non-functional improvements, e.g., improved usability, as we focused on functional aspects. In addition, some minor aspects of our solution have not been implemented yet, e.g., the GUI to specify the duration of activities.

Follow-Up Thesis – Evaluation in Industrial Projects Within this thesis, we were only capable to apply our approach to sample methods and an limited case study. Thus, future work includes the application of our approach to create more complex, real-world methods. We therefore propose to consider the evaluation and refinement of our approach within industrial projects as a possible follow-up thesis. Ideally, the approach would be applied in several projects of different size and with different project teams. Thereby, a reference method repository with method services and method patterns could be established.

Follow-Up Thesis – Integrating of Human and Organizational Aspects Lately with modern and agile software engineering methods, there is a clear trend towards team-based methods, where decisions are taken by team members instead of project managers. Most recent evolutions focus on the establishment of cross-functional teams that overcome barriers of traditionally functional oriented organization. For example, DevOps describes a culture where development and operations departments collaborate in order to establish method improvements spanning the complete lifecycle of IT systems. However, many companies struggle with introducing these novel methods due to organizational constraints and individual lack of skills.

As these recent improvements to the software engineering methodology stem from human and organizational aspects, their explicit modeling and incorporation into our approach could be investigated in a follow-up thesis. Currently, these aspects can only be expressed in roles and situational factors. Ideally, project method engineer could model these aspects and evaluate, whether the chosen project team organization fits to skills of the team members and the project situation.

With such an extended modeling approach, the transfer of our approach to other domains seems especially interesting. For example, with respect to the current research area of the fourth industrial revolution (“Industrie 4.0”), researchers investigate the impact of future industrial production processes (methods) on

human labor (project team members). Our approach could serve as a foundation to reason about the relationship of production processes and the organization and required skills of employees.

Follow-Up Thesis – Integrating Change at Runtime Our approach uses a transformation to BPEL/BPEL4People and a standard BPEL server to enable the execution of methods. Therefore, changing methods at runtime is not in scope of our approach. A possible follow-up thesis could therefore focus on enabling and handling changes to the method at runtime as described theoretically in [Gei15]. One critical aspect would include the avoidance or resolution of change-related conflicts or inconsistencies. Another challenge is the migration of the current runtime state into an instance of the changed method, possibly by adopting the models@runtime [Red+14] philosophy.

Follow-Up Thesis – Usage for Empirical Software Engineering With the increasing maturity of the software engineering field, there are more and more attempts to applied software engineering research with a strong empirical component. However, especially in the area of software engineering methods and method engineering the lack of data about method enactment is a major issue. Thus, today a lot of empirical software engineering is based on the investigation of the check-in history of open source repositories, as in [GAH16] or [SMS16].

With our approach, the enactment of methods is tightly coupled with the interaction of the project team with the BPEL server. Enactment logs are available for the BPEL server and change logs for the project repository. A follow-up thesis could therefore investigate, how our approach could be used and extended to perform empirical software engineering in order to tackle the lack of enactment data.

References

- [AHo2] Wil M. P. van der Aalst and Kees Max van Hee. *Workflow management: Models, methods, and systems*. Cambridge and Mass: MIT Press, 2002. ISBN: 0262011891.
- [AL12] Scott Ambler and Mark Lines. *Disciplined Agile Delivery: A practitioner's guide to agile software delivery in the enterprise*. Upper Saddle River and N.J.: IBM Press, 2012. ISBN: 9780132810135.
- [Amb98] Scott W. Ambler. *Process patterns: Building large-scale systems using object technology*. Cambridge, UK, and , New York: Cambridge University Press, 1998. ISBN: 9780521645683.
- [AWo5] Aybüke Aurum and Claes Wohlin. *Engineering and Managing Software Requirements*. Berlin/Heidelberg: Springer-Verlag, 2005. ISBN: 3540250433.
- [Bal07] Ricardo Balduino. *Introduction to OpenUP (Open Unified Process)*. 2007.
- [Bec00] Kent Beck. *Extreme programming eXplained: Embrace change*. XP series. Reading, MA: Addison-Wesley, 2000. ISBN: 9780201616415.
- [Bek+08] Willem Bekkers, Inge van de Weerd, Sjaak Brinkkemper, and Alain Mahieu. "The Influence of Situational Factors in Software Product Management: An Empirical Study". In: *Proceedings of the 2008 Second International Workshop on Software Product Management (ISWPM '08)*. Ed. by Christoph Ebert, Sjaak Brinkkemper, Slinger Jansen, and Gerald Heller. IEEE, 2008, pp. 41–48. ISBN: 9781424440832.
- [Ben+07] Reda Bendraou, Benoit Combemale, Xavier Cregut, and Marie-Pierre Gervais. "Definition of an Executable SPEM 2.0". In: *14th Asia-Pacific Software Engineering Conference (APSEC'07)*. Los Alamitos and Calif: IEEE Computer Society, 2007, pp. 390–397. ISBN: 0769530575.
- [Ben+10] Reda Bendraou, Jean-Marc Jezequel, Marie-Pierre Gervais, and Xavier Blanc. "A Comparison of Six UML-Based Languages for Software Process Modeling". In: *IEEE Transactions on Software Engineering* 36.5 (2010), pp. 662–675.

- [BLW96] Sjaak Brinkkemper, Kalle Lyytinen, and Richard J. Welke, eds. *Proceedings of the IFIP TC8, WG8.1/8.2 Working Conference on Method Engineering: Principles of method construction and tool support*. London: Chapman & Hall, 1996. ISBN: 041279750X.
- [BR04] Stefan Bergström and Lotta Råberg. *Adopting the Rational Unified Process: Success with the RUP*. Addison-Wesley object technology series. Boston: Addison-Wesley, 2004. ISBN: 0321202945.
- [Bre+04] Paolo Bresciani, Anna Perini, Paolo Giorgini, Fausto Giunchiglia, and John Mylopoulos. "Tropos: An Agent-Oriented Software Development Methodology". In: *Autonomous Agents and Multi-Agent Systems 8.3* (2004), pp. 203–236.
- [Bri96] Sjaak Brinkkemper. "Method engineering: engineering of information systems development methods and tools". In: *Information & Software Technology* 38.4 (1996), pp. 275–280.
- [BSHo1] Sjaak Brinkkemper, Motoshi Saeki, and Frank Harmsen. "A Method Engineering Language for the Description of Systems Development Methods". In: *Proceedings of the 13th International Conference on Advanced Information Systems Engineering (CAiSE 2001)*. Ed. by Klaus R. Dittrich, Andreas Geppert, and Moira C. Norrie. Vol. 2068. Lecture Notes in Computer Science. Berlin and Heidelberg: Springer, 2001, pp. 473–476. ISBN: 3540422153.
- [BSHg8] Sjaak Brinkkemper, Motoshi Saeki, and Anton Frank Harmsen. "Assembly Techniques for Method Engineering". In: *Proceedings of the 10th International Conference on Advanced information systems engineering (CAiSE '98)*. Ed. by Barbara Pernici and Costantino Thanos. Vol. 1413. Lecture Notes in Computer Science. Berlin: Springer Berlin / Heidelberg and Springer, 1998, pp. 381–400. ISBN: 9783540645566.
- [BT03] Barry W. Boehm and Richard Turner. "Observations on Balancing Discipline and Agility". In: *Proceedings of the Conference on Agile Development (ADC 2003)*. Los Alamitos and Calif: IEEE Computer Society, 2003, pp. 32–39. ISBN: 0769520138.
- [Cau10] Corine Cauvet. "Method Engineering: A Service-Oriented Approach". In: *Intentional Perspectives on Information Systems Engineering*. Ed. by Selmin Nurcan, Camille Salinesi, Carine Souveyet, and Jolita Ralyté. Berlin and Heidelberg: Springer Berlin Heidelberg, 2010, pp. 335–354. ISBN: 9783642125430.
- [CC05] Graham Curtis and David P. Cobham. *Business information systems: Analysis, design, and practice*. 5th ed. Harlow and England, New York: Financial Times Prentice Hall, 2005. ISBN: 9780273687924.
- [Cer+11] Mario Cervera, Manoli Albert, Victoria Torres, and Vicente Pelechano. "Turning Method Engineering Support into Reality". In: *IFIP Advances in Information and Communication Technology*. Ed. by Jolita Ralyté, Isabelle Mirbel, and Rébecca Deneckère. Vol. 351. Berlin and Heidelberg: Springer Berlin Heidelberg, 2011, pp. 138–152. ISBN: 9783642199967.

- [CH05] James O. Coplien and Neil Harrison. *Organizational patterns of agile software development*. Upper Saddle River, NJ: Pearson Prentice Hall, 2005. ISBN: 9780131467408.
- [Chro0] Gerhard Chroust. "Software Process Models: Structure and Challenges". In: *Software: theory and practice—proceedings, IFIP congress 2000*. Ed. by Yulin Feng, David S. Notkin, and M.-C Gaudel. Amsterdam: Kluwer, 2000, pp. 279–286. ISBN: 9783901882043.
- [CJ99] Reidar Conradi and Letizia Jaccheri. "Process Modelling Languages". In: *Software Process: Principles, Methodology, and Technology*. Ed. by Jean-Claude Derniame, BadaraAli Kaba, and David Wastell. Vol. 1500. Lecture notes in computer science. Springer Berlin Heidelberg, 1999, pp. 27–52. ISBN: 9783540655169.
- [CKO92] Bill Curtis, Marc I. Kellner, and Jim Over. "Process modeling". In: *Communications of the ACM* 35.9 (1992), pp. 75–90.
- [CO12] Paul Clarke and Rory V. O'Connor. "The situational factors that affect the software development process: Towards a comprehensive reference framework". In: *Inf. Softw. Technol.* 54.5 (2012), pp. 433–447.
- [Coc02] Alistair Cockburn. *Agile software development*. Addison-Wesley, 2002. ISBN: 9780201699692.
- [Coc05] Alistair Cockburn. *Crystal clear: A human-powered methodology for small teams*. The Agile software development series. Boston: Addison-Wesley, 2005. ISBN: 0201699478.
- [Coc98] Alistair Cockburn. *Surviving object-oriented projects: A manager's guide*. Reading and Mass: Addison Wesley, 1998. ISBN: 9780201498349.
- [Cop95] James O. Coplien. "A Generative Development - Process Pattern Language". In: *Pattern languages of program design*. Ed. by James O. Coplien and Douglas C. Schmidt. New York, NY, USA: ACM Press / Addison-Wesley, 1995, pp. 183–237.
- [CR06] Valentine Casey and Ita Richardson. "Uncovering the reality within virtual software teams". In: *Proceedings of the 2006 international workshop on Global software development for the practitioner*. Shanghai and China: ACM, 2006, pp. 66–72. ISBN: 1595934049.
- [CS95] James O. Coplien and Douglas C. Schmidt, eds. *Pattern languages of program design*. New York, NY, USA: ACM Press / Addison-Wesley, 1995.
- [Den+08] Rébecca Deneckère, Adrian Iacovelli, Elena Kornyshova, and Carine Souveyet. "From Method Fragments to Method Services". In: *Proceedings of the 13th International Workshop on Exploring Modeling Methods for Systems Analysis and Design (EMMSAD'08) held in conjunction with the CAiSE'08 Conference*. Ed. by Terry Halpin, Erik Proper, John Krogstie, Xavier Franch, Ela Hunt, and Remi Coletta. Vol. 337. CEUR Workshop Proceedings. 2008, pp. 80–96.

- [Dino5] Torgeir Dingsøy. "Postmortem reviews: Purpose and approaches in software engineering". In: *Information and Software Technology* 47.5 (2005), pp. 293–303.
- [DS98] Rébecca Deneckère and Carine Souveyet. "Patterns for Extending an OO Model with Temporal Features". In: *OOIS'98*. Ed. by Collete Roland and George Grosz. London: Springer London, 1998, pp. 201–218. ISBN: 9781852330460.
- [Ell+10] Ralf Ellner, Samir Al-Hilank, Johannes Drexler, Martin Jung, Detlef Kips, and Michael Philippsen. "eSPEM – A SPEM Extension for Enactable Behavior Modeling". In: *Modelling Foundations and Applications*. Ed. by Thomas Kühne, Bran Selic, Marie-Pierre Gervais, and François Terrier. Vol. 6138. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2010, pp. 116–131.
- [Ell+11] Ralf Ellner, Samir Al-Hilank, Martin Jung, Detlef Kips, and Michael Philippsen. "An Integrated Tool Chain for Software Process Modeling and Execution". In: *Joint Proceedings of co-located Events at the 8th European Conference on Modelling Foundations and Applications (ECMFA 2012)*. Ed. by Harald Störrle, Goetz Botterweck, Michel Bourdellès, Richard Paige Kolovos, Ella Roubtsova, Julia Rubin, and Juha-Pekka Tolvanen. Copenhagen and Denmark: Technical University of Denmark (DTU), 2011, pp. 73–82. ISBN: 9788764310146.
- [Eng+15] Martin Engstler, Masud Fazal-Baqaie, Eckhart Hanser, Martin Mikusz, and Alexander Volland, eds. *Projektmanagement und Vorgehensmodelle 2015: Hybride Projektstrukturen erfolgreich umsetzen- Proceedings der gemeinsamen Tagung der Fachgruppen WI-PM und WI-VM im Fachgebiet Wirtschaftsinformatik der Gesellschaft für Informatik e.V.* Vol. 250. Lecture notes in informatics. GI, Köllen Druck+Verlag GmbH, Bonn, 2015.
- [Engo8] Gregor Engels. *Quasar Enterprise: Anwendungslandschaften serviceorientiert gestalten*. 1st ed. Heidelberg: Dpunkt-Verl., 2008. ISBN: 3898645061.
- [Erl09] Thomas Erl. *SOA: Principles of service design*. 5. print. Prentice-Hall service-oriented computing series from Thomas Erl. Upper Saddle River, NJ: Prentice Hall, 2009. ISBN: 9780132344821.
- [ES10] Gregor Engels and Stefan Sauer. "A Meta-Method for Defining Software Engineering Methods". In: *Graph Transformations and Model-Driven Engineering*. Ed. by Gregor Engels, Claus Lewerentz, Wilhelm Schäfer, Andy Schürr, and Bernhard Westfechtel. Vol. 5765. Lecture Notes in Computer Science. Berlin: Springer, 2010, pp. 411–440. ISBN: 9783642173219.
- [Faz+13] Masud Fazal-Baqaie, Baris Güldali, Markus Luckey, Stefan Sauer, and Michael Spijkerman. "Maßgeschneidert und werkzeugunterstützt Entwickeln angepasster Requirements Engineering-Methoden". In: *OBJEKTSpektrum (Online Themenspecials) RE/2013* (2013), pp. 1–5.

- [FCE14] Masud Fazal-Baqaie, Christian Gerth, and Gregor Engels. "Breathing Life into Situational Software Engineering Methods". In: *In Proceedings of the 15th International Conference of Product Focused Software Development and Process Improvement (PROFES 2014)*. Ed. by A. Jedlitschka et al. Vol. 8892. Springer, 2014, pp. 281–284.
- [FE16] Masud Fazal-Baqaie and Gregor Engels. "Software Processes Management by Method Engineering with MESP". In: *Managing Software Process Evolution*. Ed. by M. Kuhrmann, A. Rausch, J. Münch, I. Richardson, and J. H. Zhang. Springer, 2016, pp. 185–210.
- [FGS15] Masud Fazal-Baqaie, Marvin Grieger, and Stefan Sauer. "Tickets without Fine - Artifact-based Synchronization of Globally Distributed Software Development in Practice". In: *Proceedings of the 16th International Conference of Product Focused Software Development and Process Improvement (PROFES 2015)*. Ed. by Pekka Abrahamsson, Luis Corral, Markku Oivo, and Barbara Russo. Springer, 2015, pp. 167–181.
- [FH02] Donald G. Firesmith and Brian Henderson-Sellers. *The OPEN process framework: An introduction*. The OPEN series. London and New York: Addison-Wesley, 2002. ISBN: 0201675102.
- [Firo9] Donald G. Firesmith. *The method framework for engineering system architectures*. Boca Raton: CRC Press, 2009. ISBN: 9781420085754.
- [FK16] Masud Fazal-Baqaie and Frank Kluthe. "Automated Quality Analysis of Software Engineering Method Models". In: *Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development (Modelward 2016)*. Ed. by Slimane Hammoudi, Luis Ferreira Pires, Bran Selic, and Philippe Desfray. Portugal: SciTePress, 2016, pp. 527–534. ISBN: 9789897581687.
- [FLE13] Masud Fazal-Baqaie, Markus Luckey, and Gregor Engels. "Assembly-Based Method Engineering with Method Patterns". In: *Software Engineering 2013 - Workshopband (inkl. Doktorandensymposium), Fachtagung des GI-Fachbereichs Softwaretechnik*. Ed. by Stefan Wagner and Horst Lichter. Vol. 215. LNI. GI, 2013, pp. 435–444. ISBN: 9783885796091.
- [FR15] Masud Fazal-Baqaie and Anu Raninen. "Successfully Initiating a Global Software Project". In: *Industrial Proceedings of the 22nd European Systems Software & Service Process Improvement & Innovation Conference (EuroSPI2015)*. Ed. by Richard Messnarz, Jorn Johansen, Morten Korsaa, Eva Christof, and Damjan Ekert. WHITEBOX, Denmark, 2015, p. 5.1.
- [FRO03] Brian Fitzgerald, Nancy L. Russo, and Tom O’Kane. "Software development method tailoring at Motorola". In: *Communications of the ACM* 46.4 (2003), pp. 64–70.
- [FSH14] Masud Fazal-Baqaie, Stefan Sauer, and Torsten Heuft. "Agile Entwicklung mit On- und Offshore-Partnern – Methodenverbesserung in der Praxis". In: *Proceedings of Projektmanagement und Vorgehensmodelle 2014*. Lecture Notes in Informatics (LNI). Bonn: GI, Köllen Druck+Verlag GmbH, 2014, pp. 59–69.

- [FSH15] Masud Fazal-Baqaie, Stefan Sauer, and Torsten Heuft. "Agile Entwicklung mit On- und Offshore-Partnern – Methodenverbesserung in der Praxis". In: *WI-MAW-Rundbrief* 21.1 (2015), pp. 5–13.
- [GAH16] Daniel M. German, Bram Adams, and Ahmed E. Hassan. "Continuously mining distributed version control systems: An empirical study of how Linux uses Git". In: *Empirical Software Engineering* 21.1 (2016), pp. 260–299.
- [Gam+07] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: Elements of reusable object oriented software*. 35. print. Addison-Wesley professional computing series. Boston, Mass. et al.: Addison-Wesley, 2007. ISBN: 9780201633610.
- [Gei15] Silke Geisen. "MAPE-K4SEM: Selbst-adaptive Software-Engineering-Methoden". PhD Thesis. Paderborn: Universität Paderborn, 2015.
- [GF15a] Marvin Grieger and Masud Fazal-Baqaie. "Towards a Framework for the Modular Construction of Situation-Specific Software Transformation Methods". In: *Softwaretechnik-Trends* 35.2 (2015), pp. 41–42.
- [GF15b] Baris Güldali and Masud Fazal-Baqaie. "Skalieren von großen agilen Projekten mit verteilten Backlogs". In: *OBJEKTSpektrum (Online Themen-specials) Agility / 2015* (2015), pp. 1–4.
- [GFS16] Marvin Grieger, Masud Fazal-Baqaie, and Stefan Sauer. "A Method Base for the Situation-Specific Development of Model-Driven Transformation Methods". In: *Softwaretechnik-Trends* (2016).
- [GGHo8] Cesar Gonzalez-Perez, Paolo Giorgini, and Brian Henderson-Sellers. "Method Construction by Goal Analysis". In: *The Inter-Networked World*. Ed. by Michael Lang, Wita Wojtkowski, Gregory Wojtkowski, Stanislaw Wrycza, and Joze Zupancic. Berlin: Springer US, 2008, pp. 79–91. ISBN: 9780387304038.
- [GHo8] Cesar Gonzalez-Perez and Brian Henderson-Sellers. "A work product pool approach to methodology specification and enactment". In: *Journal of Systems and Software* 81.8 (2008), pp. 1288–1305.
- [GJM03] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of software engineering*. 2nd ed. Upper Saddle River and N.J.: Prentice Hall, 2003. ISBN: 0133056996.
- [GJS10] Mahdi Fahmideh Gholami, Pooyan Jamshidi, and Fereidoon Shams. "A Procedure for Extracting Software Development Process Patterns". In: *Proceedings of the 4th European Modelling Symposium (EMS2010)*. Ed. by David Al-Dabass, Alessandra Orsoni, Athanasios Pantelous, Marco Vannucci, and Ajith Abraham. 2010, pp. 75–83.
- [GK09] Bastian Grabski and Lars Krüger. *Analysen zu Qualität und Qualitätsmanagement von Software und Dienstleistungen: Technical Report*. Magdeburg, Germany, 2009.
- [Gla04] Robert L. Glass. "Matching methodology to problem domain". In: *Communications of the ACM* 47.5 (2004), pp. 19–21.

- [GLS98] Göran Goldkuhl, Mikael Lind, and Ulf Seigerroth. "Method Integration: The Need For A Learning Perspective". In: *IEE Proceedings Software* 145.4 (1998), pp. 113–118.
- [GP01] Daya Gupta and Naveen Prakash. "Engineering Methods from Method Requirements Specifications". In: *Requirements Engineering* 6.3 (2001), pp. 135–160.
- [Gri+14] Marvin Grieger, Masud Fazal-Baqaie, Stefan Sauer, and Markus Klenke. "A Method to Systematically Improve the Effectiveness and Efficiency of the Semi-Automatic Migration of Legacy Systems". In: *Softwaretechnik-Trends* 34.2 (2014), pp. 77–78.
- [Gri+16] Marvin Grieger, Masud Fazal-Baqaie, Gregor Engels, and Markus Klenke. "Concept-Based Engineering of Situation-Specific Migration Methods". In: *Proceedings of the 15th International Conference on Software Reuse (ICSR2016)*. Lecture Notes in Computer Science. Springer, 2016.
- [Gro+97] Georges Grosz, Colette Rolland, Sylviane Schwer, Carine Souveyet, Veronique Plihon, Samira Si-Said, Camille Ben Achour, and Christophe Gnaho. "Modelling and engineering the requirements engineering process: An overview of the NATURE approach". In: *Requirements Engineering* 2.3 (1997), pp. 115–131.
- [Gru02] Volker Gruhn. "Process-Centered Software Engineering Environments: A Brief History and Future Challenges". In: *Annals of Software Engineering* 14.1/4 (2002), pp. 363–382.
- [Har97] Anton Frank Harmsen. "Situational Method Engineering". PhD Thesis. Twente: University of Twente, 1997.
- [HB95] Frank Harmsen and Sjaak Brinkkemper. "Design and implementation of a method base management system for a situational CASE environment". In: *Asia-Pacific Software Engineering Conference*. Los Alamitos: IEEE Computer Society, 1995, pp. 430–438.
- [HB]94] Frank Harmsen, Sjaak Brinkkemper, and J. L. Han Oei. "Situational method engineering for informational system projects". In: *Methods and Associated Tools for the Information Systems Life Cycle*. Ed. by Alex A. Verrijn-Stuart and T. William Olle. Vol. 55. IFIP Transactions. Amsterdam: North-Holland Publishers, 1994, pp. 169–194. ISBN: 0444820744.
- [Hei+10] Werner Heijstek, Michel R. V. Chaudron, Libing Qiu, and Christian C. Schouten. "A Comparison of Industrial Process Descriptions for Global Custom Software Development". In: *Proceedings of the 5th International Conference on Global Software Engineering (ICGSE)*. Los Alamitos and Calif: IEEE, 2010, pp. 277–284. ISBN: 9780769541228.
- [Hen+14] Brian Henderson-Sellers, Jolita Ralyté, Pär J. Ågerfalk, and Matti Rossi. *Situational Method Engineering*. Berlin and Heidelberg: Springer Berlin Heidelberg, 2014. ISBN: 9783642414664.
- [Heno6] Brian Henderson-Sellers. "Method Engineering: Theory and Practice". In: *Information systems technology and its applications*. Ed. by Dimitris Karagiannis. Vol. 84. Proceedings. Bonn: Ges. für Informatik, 2006, pp. 13–23. ISBN: 9783885791782.

- [HHo8] Reinhard Höhn and Stephan Höppner. *Das V-Modell XT: Anwendungen, Werkzeuge, Standards*. Springer-Lehrbuch. Berlin and Heidelberg: Springer, 2008. ISBN: 9783540302506.
- [HLHo2] Brian Henderson-Sellers, David Lowe, and Brendan Haire. "OPEN Process Support for Web Development". In: *Annals of Software Engineering* 13.1/4 (2002), pp. 163–201.
- [HMF13] Eckhart Hanser, Martin Mikusz, and Masud Fazal-Baqaie, eds. *Vorgehensmodelle 2013: Vorgehensmodelle - Anspruch und Wirklichkeit - Proceedings der 20. Tagung der Fachgruppe Vorgehensmodelle im Fachgebiet Wirtschaftsinformatik (WI-VM) der Gesellschaft für Informatik e.V.* Vol. 224. Lecture notes in informatics. Lörrach, Germany: GI, Köllen Druck+Verlag GmbH, Bonn, 2013.
- [Hoeo8] Klaus Hoermann. *Automotive SPICE in practice: Surviving interpretation and assessment*. 1st ed. Santa Barbara and Calif: Rocky Nook, 2008. ISBN: 9781933952291.
- [HR10] Brian Henderson-Sellers and Jolita Ralyté. "Situational Method Engineering: State-of-the-Art Review". In: *Journal of Universal Computer Science (J. UCS)* 16.3 (2010), pp. 424–478.
- [HRT04] Bo Hansen, Jeremy Rose, and Gitte Tjørnehøj. "Prescription, description, reflection: the shape of the software process improvement field". In: *International Journal of Information Management* 24.6 (2004), pp. 457–472.
- [HS05] Brian Henderson-Sellers and M. K. Serour. "Creating a Dual-Agility Method". In: *Journal of Database Management* 16.4 (2005), pp. 1–24.
- [Hsu+08] Nien-Lin Hsueh, Wen-Hsiang Shen, Zhi-Wei Yang, and Don-Lin Yang. "Applying UML and software simulation for process definition, verification, and validation". In: *Information and Software Technology* 50.9-10 (2008), pp. 897–911.
- [HV97] Arthur H. M. ter Hofstede and T. F. Verhoef. "On the feasibility of situational method engineering". In: *Information Systems* 22.6-7 (1997), pp. 401–422.
- [IEE90] IEEE. *Standard Glossary of Software Engineering Terminology*. New York, N.Y., USA, 1990.
- [ISO01] ISO/IEC. *Software engineering – Product quality*. Geneva and Switzerland, 2001.
- [ISO07] ISO/IEC. *Software Engineering: Metamodel for Development Methodologies*. Geneva and Switzerland, 2007.
- [ISO11] ISO/IEC. *Systems and software engineering – Systems and Software Quality Requirements and Evaluation (SQuaRE) – System and Software Quality Models*. Geneva, 2011.
- [JBD99] Letizia Jaccheri, Mario Baldi, and Monica Divitini. "Evaluating the Requirements for Software Process Modelling Languages and Systems". In: *Process support for Distributed Teambased Software Development (PDTSD'99)*, Orlando, FL. 1999, pp. 570–578.

- [JBR99] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The unified software development process*. The Addison-Wesley object technology series. Reading and Mass: Addison-Wesley, 1999. ISBN: 9780201571691.
- [KÅ05] Fredrik Karlsson and Pär J. Ågerfalk. "Method-User-Centred Method Configuration". In: *Situational Requirements Engineering Processes : Methods, Techniques and Tools to Support Situation-Specific Requirements Engineering Processes*. Ed. by Jolita Ralyté, Pär J. Ågerfalk, and N. Kraiem. Vol. 1. Irelenad: University of Limerick, 2005, pp. 37–43. ISBN: 1874653828.
- [KÅ09] Fredrik Karlsson and Pär J. Ågerfalk. "Towards Structured Flexibility in Information Systems Development: Devising a Method for Method Configuration". In: *J. Database Manag.* 20.3 (2009), pp. 51–75.
- [KÅ11] Fredrik Karlsson and Pär J. Ågerfalk. "Towards Structured Flexibility in Information Systems Development". In: *Theoretical and Practical Advances in Information Systems Development*. Ed. by Keng Siau. IGI Global, 2011, pp. 214–238. ISBN: 9781609605216.
- [KÅ12] Fredrik Karlsson and Pär J. Ågerfalk. "MC Sandbox: Devising a tool for method-user-centered method configuration". In: *Information and Software Technology* 54.5 (2012), pp. 501–516.
- [KC03] Jeffrey O. Kephart and David M. Chess. "The vision of autonomic computing". In: *Computer* 36.1 (2003), pp. 41–50.
- [KC07] Barbara Kitchenham and Stuart Charters. *Guidelines for performing Systematic Literature Reviews in Software Engineering: EBSE Technical Report EBSE-2007-01*. 2007.
- [KDS07] Elena Kornyshova, Rébecca Deneckère, and Camille Salinesi. "Method Chunks Selection by Multicriteria Techniques: an Extension of the Assembly-based Approach". In: *Situational Method Engineering: Fundamentals and Experiences*. Ed. by Jolita Ralyté, Sjaak Brinkkemper, and Brian Henderson-Sellers. Vol. 244. IFIP — The International Federation for Information Processing. Springer US, 2007, pp. 64–78. ISBN: 9780387739465.
- [KF15] Marco Kuhrmann and Daniel Mendez Fernandez. "Systematic Software Development: A State of the Practice Report from Germany". In: *IEEE 10th International Conference on Global Software Engineering (ICGSE 2015)*. 2015, pp. 51–60.
- [KFS13a] Marco Kuhrmann, Daniel Méndez Fernández, and Ragna Steenweg. "Systematic software process development: where do we stand today?" In: *Proceedings of the 2013 International Conference on Software and System Process (ICSSP 2013)*. Ed. by Jürgen Münch, Jo Ann Lan, and He Zhang. New York, NY and USA: ACM, 2013, pp. 166–170. ISBN: 9781450320627.

- [KFS13b] Marco Kuhrmann, Daniel Méndez Fernández, and Ragna Steenweg. “Systematic software process development: where do we stand today?” In: *Proceedings of the 2013 International Conference on Software and System Process (ICSSP 2013)*. Ed. by Jürgen Münch, Jo Ann Lan, and He Zhang. New York, NY and USA: ACM, 2013, pp. 166–170. ISBN: 9781450320627.
- [KKT14] Marco Kuhrmann, Georg Kalus, and Manuel Then. “The Process Enactment Tool Framework – Transformation of software process models to prepare enactment”. In: *Science of Computer Programming* 79 (2014), pp. 172–188.
- [KLR96] Steven Kelly, Kalle Lyytinen, and Matti Rossi. “MetaEdit+ A fully configurable multi-user and multi-tool CASE and CAME environment”. In: *Advanced Information Systems Engineering*. Ed. by Panos Constantopoulos, John Mylopoulos, and Yannis Vassiliou. Vol. 1080. Lecture notes in computer science. Springer Berlin Heidelberg, 1996, pp. 1–21. ISBN: 9783540612926.
- [Klu14] Frank Kluthe. *Quality Assurance of Situational Methods: Master’s thesis*. Germany, 2014.
- [KNS92] Gerhard Keller, Markus Nüttgens, and August-Wilhelm Scheer. *Semantische Prozeßmodellierung auf der Grundlage Ereignisgesteuerter Prozeßketten (EPK): Technical Report 89*. 1992.
- [Kru99] Philippe Kruchten. *The rational unified process: An introduction*. Object technology series. Reading and Mass: Addison-Wesley, 1999. ISBN: 0201604590.
- [KTF11] Marco Kuhrmann, Thomas Ternité, and Jan Friedrich. *as V-Modell® XT anpassen: Anpassung und Einführung kompakt für V-Modell® XT Prozessingenieure*. Informatik im Fokus. Berlin and Heidelberg: Springer-Verlag Berlin Heidelberg, 2011. ISBN: 9783642014901.
- [KV09] Jukka Käriäinen and Antti Välimäki. “Applying Application Lifecycle Management for the Development of Complex Systems: Experiences from the Automation Industry”. In: *Software Process Improvement*. Ed. by RoryV O’Connor, Nathan Baddoo, Juan Cuadrado Gallego, Ricardo Rejas Muslera, Kari Smolander, and Richard Messnarz. Vol. 42. Communications in Computer and Information Science. Springer Berlin Heidelberg, 2009, pp. 149–160. ISBN: 9783642041327.
- [KW04] Fredrik Karlsson and Kai Wistrand. “MC Sandbox - Tool Support for Method Configuration”. In: *CAiSE’04 Workshops in connection with The 16th Conference on Advanced Information Systems Engineering*. Ed. by Janis Grundspenkis and Marite Kirikova. Vol. 1. CAiSE workshops. Riga and Latvia: Faculty of Computer Science and Information Technology, Riga Technical University, 2004, pp. 199–210. ISBN: 9984976718.
- [KW06] Fredrik Karlsson and Kai Wistrand. “Combining method engineering with activity theory: theoretical grounding of the method component concept”. In: *European Journal of Information Systems* 15.1 (2006), pp. 82–90.

- [Loo07] Han van Loon. *Process assessment and ISO/IEC 15504: A reference book*. 2nd ed. New York: Springer, 2007. ISBN: 9780387300481.
- [Mey14] Bertrand Meyer. *Agile! The good, the hype and the ugly*. Switzerland: Springer International Publishing, 2014. ISBN: 9783319051550.
- [Mit05] Tilak Mitra. *Business-driven development: IBM developerWorks article*. Ed. by IBM. 2005.
- [MR06] Isabelle Mirbel and Jolita Ralyté. "Situational method engineering: combining assembly-based and roadmap-driven approaches". In: *Requirements Engineering 11.1* (2006), pp. 58–78.
- [Mün+12] Jürgen Münch, Ove Armbrust, Martin Kowalczyk, and Martín Soto. *Software process definition and management*. The Fraunhofer IESE Series on Software and Systems Engineering. Berlin and New York: Springer Berlin Heidelberg, 2012. ISBN: 9783642242908.
- [MW15] Daniel Méndez Fernández and Stefan Wagner. "Naming the pain in requirements engineering: A design for a global family of surveys and first results from Germany". In: *Information and Software Technology 57* (2015), pp. 616–643.
- [NC13] Anh Nguyen-Duc and D. S. Cruzes. "Coordination of Software Development Teams across Organizational Boundary - An Exploratory Study". In: *Proceedings of the 8th International Conference on Global Software Engineering (ICGSE)*. 2013, pp. 216–225.
- [Nee14] Karthik Neela. *Enactment of Software Development Methods with the Support of BPEL Workflow Engines: Master's thesis*. Germany, 2014.
- [NH03] Van Puh Nguyen and Brian Henderson-Sellers. "Towards Automated Support for Method Engineering with the Open Approach". In: *Proceedings of the Seventh IASTED International Conference on Software Engineering and Applications*. Anaheim, USA: ACTA Press, 2003, pp. 691–696.
- [OAS07] OASIS. *Web Services Business Process Execution Language Version 2.0*. 2007.
- [OAS10] OASIS. *Web Services – Human Task (WS-HumanTask) Specification Version 1.1 - Committee Draft 10 / Public Review Draft 04*. 23 June 2010.
- [OMGo8] OMG. *Software & Systems Process Engineering Metamodel Specification (SPEM) 2.0*. 2008.
- [OMG11] OMG. *Business Process Model and Notation*. 2011.
- [OMG14] OMG. *Object Constraint Language*. 2014.
- [Ost87] Leon J. Osterweil. "Software processes are software too". In: *Proceeding of the 9th International Conference on Software Engineering (ICSE)*. Washington and D.C: IEEE Computer Society Press, 1987, pp. 2–13. ISBN: 0897912160.
- [Ped+07] Oscar Pedreira, Mario Piattini, Miguel R. Luaces, and Nieves R. Brisaboa. "A systematic review of software process tailoring". In: *ACM SIGSOFT Software Engineering Notes 32.3* (2007), p. 1.

- [Per+11] Eliana Beatriz Pereira, Ricardo Melo Bastos, Michael da Costa Móra, and Toacy Cavalcante de Oliveira. "Improving the Consistency of SPEM-based Software Processes". In: *ICEIS 2011 - Proceedings of the 13th International Conference on Enterprise Information Systems (ICEIS 2011)*. Ed. by Runtong Zhang, José Cordeiro, Xuewei Li, Zhenji Zhang, and Juliang Zhang. SciTePress, 2011, pp. 76–86. ISBN: 9789898425553.
- [Pli96] Veronique Plihon. "MENTOR: An Environment Supporting the Construction of Methods". In: *Proceedings 1996 Asia-Pacific Software Engineering Conference*. Los Alamitos, CA and USA: IEEE Computer Society Press, 1996, p. 384.
- [PMo4] Vladimir Pavlov and Dmitry Malenko. "Mining MSF for Process Patterns: a SPEM-based Approach". In: *VikingPLoP 2004*. 2004, pp. 46–66.
- [PS97] Naveen Prakash and Sangeeta Sabharwal. "Building CASE Tools For Methods Represented As Abstract Data types". In: *OOIS'96*. Ed. by Dilip Patel, Yuan Sun, and Shushma Patel. Springer London, 1997, pp. 357–369. ISBN: 9783540761327.
- [Ralo4] Jolita Ralyté. "Towards situational methods for information systems development: engineering reusable method chunks". In: *Proceedings of the 13th international conference on information systems development*. Ed. by O. Vasilecas, A. Caplinskas, W. Wojtkowski, W. G. Wojtkowski, J. Zupancic, and S. Wrycza. Vilnius and Lithuania: Vilnius Gediminas Technical University, 2004, pp. 271–282.
- [Ram+06] Balasubramaniam Ramesh, Lan Cao, Kannan Mohan, and Peng Xu. "Can distributed software development be agile?" In: *Commun. ACM* 49.10 (2006), pp. 41–46.
- [RB96] Matti Rossi and Sjaak Brinkkemper. "Complexity metrics for systems development methods and techniques". In: *Information Systems* 21.2 (1996), pp. 209–227.
- [RBHo7] Jolita Ralyté, Sjaak Brinkkemper, and Brian Henderson-Sellers, eds. *Situational Method Engineering: Fundamentals and Experiences*. IFIP — The International Federation for Information Processing. Springer US, 2007. ISBN: 9780387739465.
- [RDR] Jolita Ralyté, Rébecca Deneckère, and Colette Rolland. "Towards a Generic Model for Situational Method Engineering". In: vol. 2681, p. 1029.
- [Red+14] David Redlich, Gordon Blair, Awais Rashid, Thomas Molka, and Wasif Gilani. "Research Challenges for Business Process Models at Run-Time". In: *Models@run.time*. Ed. by Nelly Bencomo, Robert B. France, Betty H.C Cheng, and Uwe Aßmann. Vol. 8378. Lecture notes in computer science. Cham: Springer International Publishing, 2014, pp. 208–236. ISBN: 9783319089140.
- [RH09] Per Runeson and Martin Höst. "Guidelines for conducting and reporting case study research in software engineering". In: *Empirical Software Engineering* 14.2 (2009), pp. 131–164.

- [RMD11] Jolita Ralyté, Isabelle Mirbel, and Rébecca Deneckère, eds. *IFIP Advances in Information and Communication Technology*. Vol. 351. Berlin and Heidelberg: Springer Berlin Heidelberg, 2011. ISBN: 9783642199967.
- [Rolo9] Colette Rolland. "Method engineering: towards methods as services". In: *Software Process: Improvement and Practice* 14.3 (2009), pp. 143–164.
- [RP96a] Colette Rolland and Veronique Plihon. "Using generic method chunks to generate process models fragments". In: *Second International Conference on Requirements Engineering*. 1996, pp. 173–180.
- [RP96b] Colette Rolland and Naveen Prakash. "A proposal for context-specific method engineering". In: *Proceedings of the IFIP TC8, WG8.1/8.2 Working Conference on Method Engineering: Principles of method construction and tool support*. Ed. by Sjaak Brinkkemper, Kalle Lyytinen, and Richard J. Welke. London: Chapman & Hall, 1996, pp. 191–208. ISBN: 041279750X.
- [RR01] Jolita Ralyté and Colette Rolland. "An Assembly Process Model for Method Engineering". In: *Proceedings of the 13th International Conference on Advanced Information Systems Engineering (CAiSE 2001)*. Ed. by Klaus R. Dittrich, Andreas Geppert, and Moira C. Norrie. Vol. 2068. Lecture Notes in Computer Science. Berlin and Heidelberg: Springer, 2001, pp. 267–283. ISBN: 3540422153.
- [RR13] Suzanne Robertson and James Robertson. *Mastering the Requirements Process: Getting Requirements Right*. 3rd ed. Upper Saddle River, NJ: Addison-Wesley Professional and Addison-Wesley, 2013. ISBN: 9780321815743.
- [SA07] Ahmed Sidky and James Arthur. "Determining the Applicability of Agile Practices to Mission and Life-Critical Systems". In: *31st IEEE Software Engineering Workshop (SEW 2007)*. 2007, pp. 3–12.
- [Sie15] Daniel Siebert. *Key Aspects of Popular Development Methods as Building Blocks for Situational Method Engineering: Master's thesis*. Germany, 2015.
- [SLS14] Andreas Spillner, Tilo Linz, and H. Schaefer. *Software testing foundations: A study guide for the certified tester exam*. 4th ed. Santa Barbara, CA: Rocky Nook, 2014. ISBN: 9781937538422.
- [SMS16] Ingo Scholtes, Pavlin Mavrodiev, and Frank Schweitzer. "From Aristotle to Ringelmann: A large-scale analysis of team productivity and coordination in Open Source Software projects". In: *Empirical Software Engineering* 21.2 (2016), pp. 642–683.
- [Sof10] Software Engineering Institute. *CMMI for Development, Version 1.3: Improving processes for developing better products and services*. Pittsburgh, Pennsylvania, 2010.
- [Som11] Ian Sommerville. *Software engineering*. 9th ed. Boston: Pearson, 2011. ISBN: 9780137035151.
- [Spi15] Michael Spijkerman. "Situationsgerechte Methodenweiterentwicklung auf Basis von MetaMe am Beispiel der Server-System-Entwicklung". PhD Thesis. Paderborn: Universität Paderborn, 2015.

- [SRG96] Samira Si-Said, Colette Rolland, and Georges Grosz. "MENTOR: A Computer Aided Requirements Engineering environment". In: *Advanced Information Systems Engineering*. Ed. by Panos Constantopoulos, John Mylopoulos, and Yannis Vassiliou. Vol. 1080. Lecture notes in computer science. Springer Berlin Heidelberg, 1996, pp. 22–43. ISBN: 9783540612926.
- [SS13] Ken Schwaber and Jeff Sutherland. *The Scrum Guide*. 2013.
- [Ste+09] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework (2nd Edition)*. 2nd ed. The eclipse series. Upper Saddle River and N.J: Addison-Wesley, 2009. ISBN: 9780321331885.
- [TR07] Samira Tasharofi and Raman Ramsin. "Process Patterns for Agile Methodologies". In: *Situational Method Engineering: Fundamentals and Experiences*. Ed. by Jolita Ralyté, Sjaak Brinkkemper, and Brian Henderson-Sellers. Vol. 244. IFIP — The International Federation for Information Processing. Springer US, 2007, pp. 222–237. ISBN: 9780387739465.
- [TRL96] Juha-Pekka Tolvanen, Matti Rossi, and Hui Liu. "Method Engineering: Current research directions and implications for future research". In: *Method Engineering*. Ed. by Sjaak Brinkkemper, Kalle Lyytinen, and Richard J. Welke. IFIP – The International Federation for Information Processing. Boston, MA: Springer US, Imprint, and Springer, 1996, pp. 296–317. ISBN: 9781475758245.
- [VB15] Leo Vijayasarathy and Charles Butler. "Choice of Software Development Methodologies - Do Project, Team and Organizational Characteristics Matter?" In: *IEEE Software* (2015), p. 1.
- [Wah+09] Ksenia Wahler, Harald C. Gall, Schahram Dustdar, and Carl-Christian Kanne. "A framework for integrated process and object life cycle modeling". PhD thesis. University of Zurich, 1.01.2009.
- [WBV07] Inge van de Weerd, Sjaak Brinkkemper, and Johan Versendaal. "Concepts for Incremental Method Evolution: Empirical Exploration and Validation in Requirements Management". In: *Advanced Information Systems Engineering*. Ed. by John Krogstie, Andreas Opdahl, and Guttorm Sindre. Vol. 4495. Lecture notes in computer science. Springer Berlin Heidelberg, 2007, pp. 469–484. ISBN: 9783540729877.
- [Wee+06] Inge van de Weerd, Sjaak Brinkkemper, Jurriaan Souer, and Johan Versendaal. "A situational implementation method for web-based content management system-applications: Method engineering and validation in practice". In: *Software Process: Improvement and Practice* 11.5 (2006), pp. 521–538.
- [Wis+00] Alexander Wise, Aaron G. Cass, Barbara Staudt Lerner, Eric K. McCall, Leon J. Osterweil, and Stanley M. Sutton. "Using Little-JIL to Coordinate Agents in Software Engineering". In: *The 15th IEEE International Conference on Automated Software Engineering*. Piscataway, N.J.: IEEE, 2000. ISBN: 0769507107.

-
- [XR08] Peng Xu and Balasubramaniam Ramesh. “Using Process Tailoring to Manage Software Development Challenges”. In: *IT Professional* 10.4 (2008), pp. 39–45.
- [ZSo7] Liming Zhu and Mark Staples. “Situational Method Quality: Situational Method Engineering: Fundamentals and Experiences”. In: *Situational Method Engineering: Fundamentals and Experiences*. Ed. by Jolita Ralyté, Sjaak Brinkkemper, and Brian Henderson-Sellers. Vol. 244. IFIP — The International Federation for Information Processing. Springer US, 2007, pp. 193–206. ISBN: 9780387739465.

Acronyms

ALM

Application Lifecycle Management.

BPEL

Business Process Execution Language.

BPMN

Business Process Model and Notation.

CASE

Computer-aided software engineering.

CDO

Connected Data Objects.

CMMI

Capability Maturity Model Integration.

DSL

Domain Specific Language.

EMF

Eclipse Modeling Framework.

EPF

Eclipse Process Framework.

GUI

graphical user interface.

MDR

method content definition requirement.

MER

method enactment requirement.

MESP

Method Engineering with Method Services and Method Patterns.

MTR

method tailoring requirement.

OCL

Object Constraint Language.

ODE

Orchestration Director Engine.

PET

Process Enactment Tool Framework.

RUP

Rational Unified Process.

SEMDM

Software Engineering Metamodel for Development Methodologies.

SME

situational method engineering.

SPEM

Software & Systems Process Engineering Meta-Model Version 2.0.

SPICE

Software Process Improvement and Capability Determination.

SR

solution requirement.

Credits

Some icons used are designed by Freepic and distributed under a free licence (with attribution).

