Universitat Politècnica de València

PhD Thesis

A Model-Driven Approach for the Design, Implementation, and Execution of Software Development Methods

Author:
Mario Cervera Úbeda

Supervisors:
Vicente Pelechano Ferragud
Manuela Albert Albiol

A thesis submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science

July 2015

Centro de Investigación en Métodos de Producción de Software

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

A Model-Driven Approach for the Design, Implementation, and Execution of Software Development Methods

This report was prepared by
Mario Cervera Úbeda

Supervisors
Dr. Vicente Pelechano Ferragud
Dra. Manuela Albert Albiol

Centro de Investigación en Métodos de Producción de Software
Universitat Politècnica de València
Camí de Vera s/n, Edif. 1F
46022 - València, Spain

Tel: (+34) 963 877 007 (Ext. 83533)
Fax: (+34) 963 877 359
Web: http://www.pros.upv.es

Release date:	16-07-2015
Comments:	A thesis submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science.
Rights:	©Mario Cervera, 2015

Acknowledgements

I can think of many people without whom this thesis would not have been written. To all of these people, I am greatly indebted. Here, I take the opportunity to extend my sincere thanks to all of them.

First of all, I would like to thank my supervisors, Dr. Vicente Pelechano and Dra. Manuela Albert, for giving me the opportunity to undertake this fascinating work. I have learned many valuable lessons during the long journey that the PhD represents. Your advice, reviews, and discussions, together with the help of Dra. Victoria Torres, have contributed to improve the work that is presented in this thesis. I am grateful to the three of you.

A big gratitude is also due to my current and former colleagues from the PROS research center for the many good moments that we have shared together. Thank you so much, Clara and María. Your encouragement has always been the main source of my strength and determination to finish my PhD. Without your support, this thesis would not have been possible. I also want to give special thanks to Marce, Vero, Sergio, Diego, Ignacio, Mariajo, Paco, and all my friends from laboratory 1L04. All of you gave me the best experiences of my last years as a PhD student. Due to these experiences, I have always been able to keep my mind clear and focused. I am also very grateful to Caro. Your stay in Valencia was the happiest time of my PhD. I will never forget that, and I will always miss you. Thanks also to Salva, for the great padel matches that we played together. Thanks Pau for being my friend, and, at the same time, my greatest inspiration. I am also thankful to my "labmates" (and "groupmates") María O., Pablo, Isma, Nacho, Miriam, Pedro, and Joan. All of you helped me grow as a researcher. Thanks Óscar for your great direction of our research center, and Ana for the valuable time that you saved me with your administrative work.

In the work environment, not only people from university have contributed to bring this thesis to fruition. Thanks Begoña Bonet for trusting in my work from the beginning. Your professional support and your deep expertise have been a great source of inspiration for me. I am also in debt to the MOSKitt team – Marc, Miguel, Héctor, Javi, and Gabi – and, especially, to the project leader, Javier Muñoz. The years that I spent with all of you are the origin of this thesis. I sincerely thank you for that.

Outside the workplace, I can mention many people that have made the PhD a more pleasant journey. Foremost, my family; especially, my parents Fernando and M^{a} Carmen, my brothers Óscar and Fer, and my sister Norma. I will be forever thankful for the love and motivation that you have given me all these years, and all my life. Thanks Fer also for your useful reviews and comments on the last versions of the thesis. I also want to express my gratitude to my friends, from Valencia and Navajas, for always being there when I needed you. Among these friends, I cannot forget to mention my turkish friend Gamze. Thanks for your unlimited patience with me, for always being so cheerful and happy, and, above all, for your friendship. Your departure saddens me deeply, but I am very grateful to you for making my last year of PhD a more enjoyable experience.

Finally, I want to thank the *Universitat Politècnica de València* and the *Conselleria de Infraestructuras, Territorio y Medio Ambiente* for the research fellowships that provided the economic resources needed to make possible the development of this thesis.

Abstract

Software development projects are diverse in nature. For this reason, software companies are often forced to define their methods in-house. In order to define methods efficiently and effectively, software companies require systematic solutions that are built upon sound methodical foundations. Providing these solutions is the main goal of the Method Engineering discipline.

Method Engineering is the discipline to design, construct, and adapt methods, techniques, and tools for the development of information systems. Over the last two decades, a lot of research work has been performed in this area. However, despite its potential benefits, Method Engineering is not widely used in industrial settings. Some of the causes of this reality are the high theoretical complexity of Method Engineering and the lack of adequate software support.

In this thesis, we aim to mitigate some of the problems that affect Method Engineering by providing a novel methodological approach that is built upon Model-Driven Engineering (MDE) foundations. The use of MDE enables a rise in abstraction, automation, and reuse that allows us to alleviate the complexity of our Method Engineering approach. Furthermore, by leveraging MDE techniques (such as metamodeling, model transformations, and models at runtime), our approach supports three phases of the Method Engineering lifecycle: design, implementation, and execution. This is unlike traditional Method Engineering approaches, which, in general, only support one of these phases.

In order to provide software support for our proposal, we developed a Computer-Aided Method Engineering (CAME) environment that is called MOSKitt4ME. To ensure that MOSKitt4ME offered the necessary functionality, we identified a set of functional requirements prior to developing the tool. Then, after these requirements were identified, we defined the architecture of our CAME environment, and, finally, we implemented the architecture in the context of Eclipse.

The thesis work was evaluated by means of a study that involved the participation of end users. In this study, MOSKitt4ME was assessed by means of the Technology Acceptance Model (TAM) and the Think Aloud method. While the TAM allowed us to measure usefulness and ease of use in a subjective manner, the Think Aloud method allowed us to analyze these measures objectively. Overall, the results were favorable. MOSKitt4ME was highly rated in perceived usefulness and ease of use; we also obtained positive results with respect to the users' actual performance and the difficulty experienced.

Resumen

Los proyectos de desarrollo de software son diversos por naturaleza. Por este motivo, las compañías de software se ven forzadas frecuentemente a definir sus métodos de manera interna. Para poder definir métodos de forma efectiva y eficiente, las compañías necesitan soluciones sistemáticas que estén definidas sobre unos fundamentos metodológicos sólidos. Proporcionar estas soluciones es el principal objetivo de la Ingeniería de Métodos.

La Ingeniería de Métodos es la disciplina que aborda el diseño, la construcción y la adaptación de métodos, técnicas y herramientas para el desarrollo de sistemas de información. Durante las dos últimas décadas, se ha llevado a cabo mucho trabajo de investigación en esta área. Sin embargo, pese a sus potenciales beneficios, la Ingeniería de Métodos no se aplica ampliamente en contextos industriales. Algunas de las principales causas de esta situación son la alta complejidad teórica de la Ingeniería de Métodos y la falta de un apropiado soporte software.

En esta tesis, pretendemos mitigar algunos de los problemas que afectan a la Ingeniería de Métodos proporcionando una propuesta metodológica innovadora que está basada en la Ingeniería Dirigida por Modelos (MDE). El uso de MDE permite elevar el nivel de abstracción, automatización y reuso, lo que posibilita una reducción de la complejidad de nuestra propuesta. Además, aprovechando técnicas de MDE (como por ejemplo el metamodelado, las transformaciones de modelos y los modelos en tiempo de ejecución), nuestra aproximación da soporte a tres fases del ciclo de vida de la Ingeniería de Métodos: diseño, implementación y ejecución. Esto es a diferencia de las propuestas existentes, las cuales, por lo general, sólo dan soporte a una de estas fases.

Con el objetivo de proporcionar soporte software para nuestra propuesta, implementamos una herramienta CAME (Computer-Aided Method Engineering) llamada MOSKitt4ME. Para garantizar que MOSKitt4ME proporcionaba la funcionalidad necesaria, definimos un conjunto de requisitos funcionales como paso previo al desarrollo de la herramienta. Tras la defineción de estos requisitos, definimos la arquitectura de la herramienta CAME y, finalmente, implementamos la arquitectura en el contexto de Eclipse.

El trabajo desarrollado en esta tesis se evaluó por medio de un estudio donde participaron usuarios finales. En este estudio, MOSKitt4ME se evaluó por medio del Technology Acceptance Model (TAM) y del método Think Aloud. Mientras que el TAM permitió medir utilidad y facilidad de uso de forma subjetiva, el método Think Aloud permitió analizar estas medidas objetivamente. En general, los resultados obtenidos fueron favorables. MOSKitt4ME fue valorado de forma positiva en cuanto a utilidad y facilidad de uso percibida; además, obtuvimos resultados positivos en cuanto al rendimiento objetivo de los usuarios y la dificultad experimentada.

Resum

Els projectes de desenvolupament de programari són diversos per naturalesa. Per aquest motiu, les companyies es veuen forçades freqüenment a definir els seus mètodes de manera interna. Per poder definir mètodes de forma efectiva i eficient, les companyies necessiten solucions sistemàtiques que estiguin definides sobre uns fundaments metodològics sòlids. Proporcionar aquestes solucions és el principal objectiu de l'Enginyeria de Mètodes.

L'Enginyeria de Mètodes és la disciplina que aborda el diseny, la construcció i l'adaptació de mètodes, tècniques i eines per al desenvolupament de sistemes d'informació. Durant les dues últimes dècades, s'ha dut a terme molt de treball de recerca en aquesta àrea. No obstant, malgrat els seus potencials beneficis, l'Enginyeria de Mètodes no s'aplica àmpliament en contextes industrials. Algunes de les principals causes d'aquesta situació són l'alta complexitat teòrica de l'Enginyeria de Mètodes i la falta d'un apropiat suport de programari.

En aquesta tesi, pretenem mitigar alguns dels problemes que afecten a l'Enginyeria de Mètodes proporcionant una proposta metodològica innovadora que està basada en l'Enginyeria Dirigida per Models (MDE). L'ús de MDE ens permet elevar el nivell d'abstracció, automatització i reutilització, possibilitant una reducció de la complexitat de la nostra proposta. A més a més, aprofitant tècniques de MDE (com per exemple el metamodelat, les transformacions de models i els models en temps d'execució), la nostra aproximació suporta tres fases del cicle de vida de l'Enginyeria de Mètodes: diseny, implementació i execució. Açò és a diferència de les propostes existents, les quals, en general, només suporten una d'aquestes fases.

Amb l'objectiu de proporcionar suport de programari per a la nostra proposta, implementàrem una eina CAME (Computer-Aided Method Engineering) anomenada MOSKitt4ME. Per garantir que MOSKitt4ME oferia la funcionalitat necessària, definírem un conjunt de requisits funcionals com a pas previ al desenvolupament de l'eina. Després de la definició d'aquests requisits, definírem la arquitectura de l'eina CAME i, finalment, implementàrem l'arquitectura en el contexte d'Eclipse.

El treball desenvolupat en aquesta tesi es va avaluar per mitjà d'un estudi on van participar usuaris finals. En aquest estudi, MOSKitt4ME es va avaluar per mitjà del Technology Acceptance Model (TAM) i el mètode Think Aloud. Mentre que el TAM va permetre mesurar utilitat i facilitat d'ús de manera subjectiva, el mètode Think Aloud va permetre analitzar aquestes mesures objectivament. En general, els resultats obtinguts van ser favorables. MOSKitt4ME va ser valorat de forma positiva pel que fa a utilitat i facilitat d'ús percebuda; a més a més, vam obtenir resultats positius pel que fa al rendiment objectiu dels usuaris i a la dificultat experimentada.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software development projects are diverse in nature. They differ, for example, in size, application domain, or expertise of the development team. Due to these differences, it is now generally recognized that software companies must define their methods in-house [1, 2, 3, 4]; thus, methods can be adapted to the needs of specific projects. To define methods efficiently and effectively, software companies require systematic solutions that are built upon sound methodical foundations. Providing these solutions is the main goal of the Method Engineering discipline.

Method Engineering is defined as "the engineering discipline to design, construct, and adapt methods, techniques, and tools for the development of information systems" [5]. Over the last two decades, a lot of research work has been performed in the Method Engineering field (see e.g. [6, 7, 8, 9]). However, even though all of this work has established a solid theoretical basis, Method Engineering is not widely used in industrial settings [3, 6, 10, 11]. This is because the costs of applying Method Engineering are usually perceived as being larger than those of using out-of-the-box methods, which are delivered by large vendors and consulting companies [3]. Examples of these methods are those for agile development (e.g., eXtreme Programming [12], Scrum [13], and Crystal Clear [14]), iterative approaches (such as the Rational Unified Process [15] and the Spiral model [16]) or generic frameworks (such as the Microsoft Solutions Framework [17]). Some of the reasons why companies tend toward these "pre-packaged" methods (rather than building project-specific methods via Method Engineering) are the high theoretical complexity of Method Engineering and the lack of adequate software support.

In this thesis, we aim to alleviate some of the problems that affect Method Engineering by providing a novel methodological approach (as well as a supporting software infrastructure) that is built upon Model-Driven Engineering (MDE) foundations. By leveraging MDE in the context of Method Engineering, we aim to facilitate the application of our proposal, and, hence, foster its adoption in industrial settings. Specifically, the proposal makes intensive use of MDE techniques (such as metamodeling, model transformations, and models at runtime) to support three phases of the Method Engineering lifecycle: design, implementation, and execution. Other phases, such as the specification of method requirements, are out of the scope of this thesis.

Method Design. The design of a software development method involves the conceptual definition of all the elements that comprise the method, such as the tasks to be carried out by software engineers, the roles that participate in these tasks, and the products to be developed to reach the final system. The method definition is built according to a method specification language, for instance the SPEM 2.0 standard [18].

Method Implementation. The method implementation involves the construction of a software system that provides complete support for the method. This system allows developers to perform Software Engineering activities; for this reason, we use the term Computer-Aided Software Engineering (CASE) to denote the system that is built in the method implementation phase. According to Fuggetta's classification of CASE technology [19], this thesis deals with *environments* (more specifically, *integrated* and *process-centered* environments); thus, we use as synonyms terms such as CASE environment, integrated environment, or simply software environment.

Method Execution. The method execution involves the enactment of method instances in specific software development projects. This enactment is tool-assisted; that is, the integrated environment that is obtained in the implementation phase makes use of the method specification to guide software engineers during the entire process of software development.

In addition to these three phases, the Method Engineering approach that is proposed in this thesis also supports the two main parts that generally comprise methods: product and process.

Product. The product part of a software development method defines the artifacts to be produced and/or consumed during the method execution as well as the tools (e.g., textual/graphical editors and code generators) that enable the creation and manipulation of these artifacts. The products of a method can be internally oriented (i.e., they are only used by people directly involved in the development process), but they can also be delivered to the customer. Examples of method products are models, code, and technical reports [20].

Process. The process part of methods is understood in two slightly different ways. Some authors consider the process part as the overall development process of the method, which encompasses all the task-related issues that are needed for software development (e.g., the tasks to be carried out until the final software system is delivered, the participants that are involved in these tasks, and the workflow that establishes the task execution order) [21, 22]. In contrast, other authors use the term process at a smaller scale, considering a process as a "product-producer" (i.e., as the description of how a single method product must be built) [4, 7, 20, 23]. In this thesis, we consider processes at the greater scale, and, hence, we denote hereafter the overall process of methods as the method process part. We use the term guidance to denote the micro-processes that specify how to develop single method products.

In addition to the product and process parts, some authors highlight other aspects that must be taken into consideration: the method tools [24, 25] and the method participants, which are also referred to as producers, actors, roles, or people [3, 26]. Our approach supports these two aspects but we consider them to be conceptually related to the method product and process parts; specifically, we consider the method tools to be part of the product aspects and the method participants to be part of the process aspects.

The rest of this chapter is organized as follows. First, Section 1.1 explains the motivations for this work. Then, Section 1.2 states the problem that the thesis addresses and Section 1.3 summarizes how the thesis contributes to solve this problem. Section 1.4 introduces the research method that has been followed. Finally, Section 1.5 explains the thesis context and Section 1.6 outlines the thesis structure.

1.1 Research Motivation

Software development methods are systematic approaches for building software, based on a particular way of thinking, which can govern the disciplined execution of software development projects (adapted from [3]). Thus, methods define *what* to do, *how*, and *when*, contributing to a better understanding of the problem, and, therefore, to an improvement in quality of the developed software [27].

Even though various attempts have been made to develop methods that are adequate for all situations (e.g., eXtreme Programming [12], Scrum [13], and the Rational Unified Process [15]), industrial projects have demonstrated that methods must be adapted to context needs [2, 3, 28]. In order to properly obtain project-specific methods, solutions that enable the efficient construction and adaptation of methods need to be sought. The Method Engineering discipline emerged as the most promising alternative to meet this need.

Method Engineering is concerned with the design of methods and the construction of the software tools that support their execution. Thus, method engineers are in charge of producing method specifications and customizing CASE environments so that software engineers can make use of these methods and tools to produce software systems [29]. In other words, method engineers facilitate the task of software engineers by providing customized methods and tools for software development.

In the literature, we can find Method Engineering approaches of different nature. Most of these approaches are assembly-based; that is, they promote the creation of reusable method parts [30], which are later assembled to obtain methods that are adapted to the context of use [7, 8, 23]. The main benefits of assembly-based Method Engineering are increased reuse, modular methods, systematized method construction, and the possibility to build repositories of method knowledge.

Regardless of its potential benefits, Method Engineering has never been widely practiced in industrial settings [6, 10]. Kuhrmann *et al.* concluded in a recent mapping study [11] that there are hardly any reports on the practical application of Method Engineering available in the literature. Henderson-Sellers *et al.* argue in [3, 31] that practitioners often fail to see the usefulness of Method Engineering mainly due to its complexity and cost in terms of time, money, and people. The complexity of Method Engineering was also noted by Ter Hofstede *et al.* [28], who identified several complexity issues related to the selection, storage, retrieval, and assembly of method fragments.

Due to the complexity of Method Engineering, practitioners often advocate the use of out-of-the-box methods since they are presented as ready for immediate use. However, using an out-of-the-box method has one important drawback: the method will most likely be an inappropriate description of the company's actual way of working. This drawback has two mayor consequences. First, it brings extra costs to the company since the method has to be adapted to the company's needs [2, 3, 32, 33]. Second, it decreases method use: developers will tend to avoid the method due to their lack of familiarity with the prescribed activities and workflows [6, 34, 35].

Software companies can alleviate these problems by adopting Method Engineering; thus, companies gain flexibility to build project-specific methods, and, since these methods are defined in-house, developers are motivated to use them due to the feeling of method ownership [36]. Nonetheless, to reap these benefits, companies need automated software support that facilitates Method Engineering activities.

The software tools that support Method Engineering are called Computer-Aided Method Engineering (CAME) environments [37, 38, 39, 40]. In their ideal form, these tools support the entire lifecycle of Method Engineering (from the initial design of the method, through its implementation, to the final method execution) as well as the two parts that generally comprise methods (product and process). However, despite the great achievements of existing CAME environments (e.g., the design of the product part of methods is fully supported by a wide variety of metamodeling notations [41]), these tools still present important deficiencies: incomplete support to the Method Engineering lifecycle, weak process support, and high complexity; furthermore, these tools are affected by the negative perception of practitioners with respect to the usefulness of Method Engineering. We argue that all of these problems are the key determinants of the little use of Method Engineering in industrial contexts.

In view of the above problems, we argue that new methodological approaches and software tools for Method Engineering must be provided. In this thesis, we face this challenge. Specifically, we contribute to the Method Engineering field by providing a new methodological approach that systematizes the design, implementation, and execution of software development methods. Our approach differs from traditional Method Engineering in that it is lightweight: it is built upon reusability principles and it is also model-driven, which enables a high level of automation. We have also developed a software environment that implements the proposal and avoids the main deficiencies of current CAME technology. Even though the development of this kind of tools is far from trivial, the use of MDE techniques has been a good means to properly handle the inherent complexity of Method Engineering.

1.2 Problem Statement

Method Engineering is an open research area. The above discussion indicates that a large number of approaches have been proposed to deal with software development methods in different ways; nonetheless, there are still important issues that need to be improved so that Method Engineering techniques can be efficiently applied in practice. The most significant problems that we have identified are the following:

High complexity. Existing Method Engineering theory is perceived as being highly complex, and, hence, most software companies advocate the use of out-of-the-box methods and CASE environments. The major problem is that these methods and tools are not adapted to the context of use, and, usually, they do not meet all software engineers needs. Thus, development teams may end up modifying their way of working according to the selected methods and tools (or just ignoring them), instead of adapting the method and tools to their needs.

Incomplete support to the lifecycle of methods. In the literature, we can find Method Engineering approaches that define precise engineering solutions to carry out the design of methods; however, it is not yet clear how the method specifications can be leveraged for the systematic construction of the supporting software environments, and also how the method specifications can be executed by these tools.

Weak process support. While the product part of methods is fully considered by most Method Engineering approaches, the process part is less well-supported. The main cause of this reality is that the ultimate goal of software development methods is the construction of a software product. Thus, the Method Engineering community has mainly concentrated on the intermediate products that are necessary to reach such final products, and also on the guidelines that assist in the creation of these intermediate products.

Limited software support. None of the existing proposals has been successful in providing a software infrastructure that allows method engineers to carry out the design, implementation, and execution of software development methods. This reality is shown in a recent study on CAME technology [41], which concludes that existing CAME (and metaCASE) environments are incomplete prototypes that only cover part of the Method Engineering lifecycle. This is one of the reasons why these tools have not achieved the expected industrial success and only MetaEdit+ [40, 42] has traversed academic boundaries.

The objective of this work is to address all of these problems. To this end, we defined four research questions that guided the research that was performed in this thesis. Each of these questions is related to one of the identified problems.

Research Question 1 (RQ1). What techniques can help alleviate the inherent complexity of Method Engineering?

Research Question 2 (RQ2). How to bridge the gap between the conceptual design of methods and the technical details of the method implementation and execution?

Research Question 3 (RQ3). How to provide process support in a Method Engineering context?

Research Question 4 (RQ4). What are the requirements of a software infrastructure that supports the design, implementation, and execution of methods?

These research questions are analyzed and answered in Section 1.3 in order to illustrate the contributions of this thesis.

1.3 Thesis Contributions

The major contribution of this thesis is a methodological approach that allows method engineers to design methods and also to build integrated software environments that support the method execution. The proposed methodological approach has been designed to answer the research questions that are formulated in Section 1.2.

RQ1. What techniques can help alleviate the inherent complexity of Method Engineering?

The answer to this question lies in the many successful techniques that are applied in MDE, such as raising the abstraction level by hiding platform-specific details, taking advantage of models to improve communication, and taking advantage of model transformations to automate repetitive work, improve software quality, and promote reuse [43]. Of all the techniques that are applied in MDE, the rise in abstraction, automation, and reuse plays the most important role in reducing Method Engineering complexity. In Software Engineering, abstraction represents one of the fundamental principles for reducing complexity by means of the removal of unnecessary details [44]; automation frees software engineers from having to perform potentially complex and error-prone tasks; and reusability minimizes workload by enabling rapid software development through the composition of reusable assets. We apply all of these ideas in the context of Method Engineering; thus, our methodological approach makes intensive use of MDE techniques to support the design, implementation, and execution of methods. Specifically, in our approach, methods are designed as models and method implementations are automatically generated from these models via model transformations. We also leverage models at runtime to reduce the complexity of the method execution; thus, the modeling effort that is made at design time is not only useful for producing CASE environments, but it also assists software engineers during the entire process of Software Engineering.

In order to assess the actual reduction in Method Engineering complexity, we performed an evaluation study that put our methodological approach to practical use (by means of the software environment that implements the proposal). To measure the subjective perception of the subjects of the study, we used the Technology Acceptance Model (TAM) [45], which allowed us to assess perceived usefulness and ease of use[1]. To reinforce the subjective results of the TAM, we also evaluated usefulness and ease of use in an objective manner. To do this, we analyzed the subjects' actual improvement in performance and the difficulties that they experienced during the study[2]. Performance was assessed by measuring efficiency and effectiveness. Difficulty was assessed by analyzing the subjects' reasoning processes, which reveal (among other data) the errors made by the subjects and the doubts that they experienced. To analyze this data at the highest possible level of detail, we applied the Think Aloud method [48]. Overall, the results of the study were favorable. Our positive results contrast with traditional Method Engineering, whose usefulness is often negatively perceived by practitioners and whose complexity remains an unsolved issue.

RQ2. How to bridge the gap between the conceptual design of methods and the technical details of the method implementation and execution?

In this thesis, we study the linkage between the design, implementation, and execution of software development methods. Current Method Engineering approaches provide advanced engineering techniques to carry out the design of methods; however, most of these approaches do not take advantage of method models to obtain the supporting software environments in a systematic manner. To overcome this limitation, in this thesis we use method models as input of model transformations that automate the construction of CASE environments that support the methods; we also illustrate the feasibility of using method models (which are created at design time) during runtime for driving the behavior of these CASE environments. By using method models during the method implementation and execution phases

[1]We consider ease of use to be the opposite of complexity [46, 47].

[2]According to Davis [45], perceived usefulness and perceived ease of use are the people's subjective appraisal of performance and effort/difficulty, respectively.

(and not only during method design), we leverage method models going one step further than state-of-the-art Method Engineering.

In order to enable the automatic construction of software environments directly from method models, we need to bridge the gap between the high-level concepts that are used at method design and the technical details of the method implementation and execution. To this end, we encapsulate in reusable assets the tools to be integrated in the software environments. Then, method engineers can associate these assets to conceptual elements of the method models. A reusable asset that is associated to a method element indicates the tool that will support the element during the method execution; for instance, a UML editor that is associated to a product called “Class model” will support the creation of specific instances of this product (i.e., specific UML class models).

RQ3. How to provide process support in a Method Engineering context?

We show how Process Modeling Languages can be used in the context of Method Engineering to achieve two major goals: (1) supporting the specification of the process part of methods during the method design phase of the Method Engineering lifecycle, and (2) enabling the execution of this process using the integrated environment that is obtained in the method implementation phase. In order to achieve the first goal without neglecting product support, our approach defines a Domain-Specific Language (DSL) that combines concepts from two standard Process Modeling Languages: SPEM 2.0 [18] and BPMN 2.0 [49]. We advocate the combination of these languages because SPEM 2.0 provides suitable primitives for method modeling, while the concepts of BPMN 2.0 enable the specification of more complex processes (e.g., processes that define branching conditions and synchronizations, which are not supported by SPEM 2.0). To meet the second goal, it is necessary that the process part of the methods that are defined using our DSL is executable. To satisfy this requirement, we implemented a model transformation that takes a method model as input and automatically obtains an executable representation of the method process part. The process model that is obtained as output of the transformation is compliant with BPMN 2.0; thus, we ensure that the model can be executed by process engines that fully support and interpret the operational semantics of BPMN 2.0 [49].

In order to evaluate the extent to which SPEM 2.0 and BPMN 2.0 are suitable languages for providing process support in the context of Method Engineering, we performed an in-depth analysis of these two languages. To this end, we applied the evaluation framework that Niknafs *et al.* present in [21]. This framework consists of a set of quality criteria for evaluating Process Modeling Languages with respect to their suitability for Method Engineering. We obtained positive results in our analysis of SPEM 2.0 and BPMN 2.0. These results are detailed in Appendix A.

RQ4. What are the requirements of a software infrastructure that supports the design, implementation, and execution of methods?

With the aim of providing software support for our proposal, in this thesis we identify a set of functional requirements to be met by a CAME environment that supports the three phases of our methodological approach. Based on these requirements, we define a technology-independent architecture and provide implementation details of this architecture in the context of Eclipse[3], more specifically in the context of the MOSKitt platform[4]. The implementation of the architecture resulted in a CAME environment that is called MOSKitt4ME. This CAME environment is ready for download at `http://users.dsic.upv.es/~mcervera/moskitt4me`.

It is important to note that the development of a CAME environment that supports our methodological approach demonstrates the feasibility of the approach. This is in accordance with one of the principles of design science, which states that the instantiation of a design artifact demonstrates its feasibility [50] (e.g., implementing a system that automates a process demonstrates that the process can, in fact, be automated). This is called "proof by construction". In the context of this thesis, the implementation of our approach as a CAME environment demonstrates that it is possible to systematize (and partially automate) the design, implementation, and execution of methods. Furthermore, the fact that our CAME environment supports these three phases validates the requirements that were identified to develop the architecture of the tool.

[3] http://www.eclipse.org/
[4] http://www.moskitt.org/

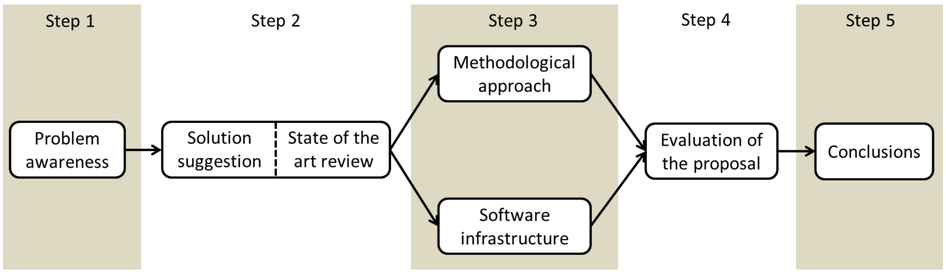


FIGURE 1.1: Research method followed in this thesis

1.4 Research Method

In order to perform the work of this thesis, we carried out a research project following the method for performing design research in information systems that is described in [51, 52]. Design research involves the analysis of the use and performance of designed artifacts to understand, explain, and, very frequently, to improve the behavior of information systems.

As Figure 1.1 shows, the design cycle consists of five steps: (1) awareness of the problem, (2) solution suggestion, (3) development, (4) evaluation, and (5) conclusion. This cycle is an iterative process: the knowledge produced by constructing and evaluating new artifacts can be used as input for a better awareness of the problem. Following the process shown in Figure 1.1, we started with the awareness of the problem; specifically, we identified the problem to be solved and we stated it clearly. Next, we performed the second step; this step involved the suggestion of a solution to the problem and the analysis of the improvements that this solution introduces with respect to already existing solutions. To do this, we studied in detail the most relevant approaches from similar domains. Once the solution to the problem was described, we developed it (step 3). This step involved two tasks. First, we defined a methodological approach that covers the design, implementation, and execution of methods. Second, we defined and implemented a software infrastructure that supports the proposed approach. When the approach and its supporting software infrastructure were completely developed, we evaluated them (step 4). Finally, we analyzed the results of our research work in order to draw several conclusions as well as to delimit areas for further research (step 5).

1.5 Context of the Thesis

This thesis has been developed in the research center *Centro de Investigación en Métodos de Producción de Software* (PROS) of the *Universitat Politècnica de València*. More specifically, the solutions proposed in this work have been defined and implemented within the context of the MOSKitt project.

The MOSKitt project constitutes a jointly work between the PROS and the Valencian Regional Ministry of Infrastructure, Territory, and Environment (also known as CITMA). This project started in 2007 and its main goal was to develop an Eclipse-based CASE environment that, making intensive use of MDE principles, provides software support to gvMétrica[5]: the software development method used at the CITMA. There is a big community involved in the MOSKitt project, ranging from analysts (software and business analysts) to end users, which are in charge of validating each new release of the tool. This setting has constituted an adequate environment to validate the approach proposed in this thesis since we successfully applied it to design, implement, and execute the gvMétrica method.

Additionally, this thesis has been developed with the support of the MICINN and the ITEA2 programme under the following research projects:

EVERYWARE. Construcción de software adaptativo para la integración de personas, servicios y cosas usando modelos en tiempo de ejecución. CICYT project referenced as TIN2010-18011.

OPEES. Open Platform for the Engineering of Embedded Systems[6]. ITEA2 project referenced as TSI-020400-2010-36.

1.6 Outline

This thesis is structured in seven chapters (including the present chapter) and three appendices. As a guide for the reader, below we briefly describe each of these chapters and appendices:

[5] http://www.gvpontis.gva.es/cast/proyectos-integra/
[6] http://www.opees.org/

- **Chapter 2** provides the reader with the knowledge that is required for understanding the overall thesis work. This chapter is divided into two main parts. The first part presents the background information that is required for understanding the methodological approach that is proposed in this thesis; the second part introduces the technological context in which the supporting software infrastructure was developed.

- **Chapter 3** summarizes the main research efforts that have been carried out in Method Engineering – the domain that is the focus of the present work. From this summary, we extract common limitations of existing Method Engineering approaches. These limitations allow us to disclose the main contributions of this thesis.

- **Chapter 4** thoroughly details the methodological approach that is proposed in this thesis. This methodological approach aims to mitigate the problems that are identified in Chapter 3.

- **Chapter 5** presents MOSKitt4ME, the software infrastructure that has been developed to support the methodological approach that is detailed in Chapter 4. Chapter 5 presents the architecture of MOSKitt4ME first; then, it details the MOSKitt4ME functionality.

- **Chapter 6** presents an evaluation study that evaluates MOSKitt4ME in terms of complexity, one of the major handicaps of existing Method Engineering approaches. The main goal of this study is to illustrate that MOSKitt4ME can be positively rated in terms of perceived usefulness and perceived ease of use, and that it can also improve the users' performance while posing little difficulty of use.

- **Chapter 7** draws some conclusions about the present thesis and it also summarizes the main results and publications that we obtained. This chapter also discusses future research directions, which are in line with the limitations of our work.

- **Appendix A** presents a comparative analysis between the SPEM 2.0 and BPMN 2.0 standards. This analysis represents the rationale behind some of the decisions that were made when we developed the methodological approach that is presented in this thesis.

- **Appendix B** presents a case study, which aims to exemplify our model-driven Method Engineering approach. To this end, the appendix applies the approach to an example method. This method represents an excerpt of the software development method that was defined by the Valencian Regional Ministry of Infrastructure, Territory, and Environment: gvMétrica.

- **Appendix C** includes material that was used during the evaluation study that is presented in Chapter 6.

Chapter 2

Background and Technological Context

This chapter provides the knowledge that is required for understanding the overall thesis work. In line with the objective of the thesis – to provide (1) a methodological approach and (2) a software infrastructure to support model-driven Method Engineering – this chapter is divided into two main parts. First, the chapter presents the background information that is required for understanding the methodological approach. Second, the chapter introduces the technological context in which the software infrastructure was developed.

The first two sections (2.1 and 2.2) focus on the thesis **background**. Specifically, Section 2.1 describes the particular domain that is the focus of our methodological approach: the Method Engineering field. In this section, we explain the basic concepts of the discipline as well as the different phases that comprise the Method Engineering lifecycle. Then, Section 2.2 introduces the development paradigm that lays the foundations of our approach: Model-Driven Engineering (MDE).

The next section (2.3) focuses on the **technological context**. Specifically, this section describes the eclipse-based technologies involved in the development of the software infrastructure that is provided in this thesis. These technologies comprise, among others, Eclipse modeling tools[1] (such as EMF, GMF, and XText) and MOSKitt, which is the platform that establishes the basis of our software infrastructure. Finally, Section 2.4 concludes the chapter.

2.1 Method Engineering

The term Method Engineering was first introduced during the mid-eighties by Bergstra *et al.* in [53]. After that, Kumar *et al.* [54] named it Methodology Engineering, but Brinkkemper *et al.* [5, 55] recommended changing it to Method Engineering, a term that has been generally accepted since. From the inception of the Method Engineering discipline, many research efforts have attempted to provide solutions to the challenges that Method Engineering entails. A state-of-the-art review of the most relevant contributions has been published in [3].

2.1.1 Defining Method Engineering

Method Engineering is defined as “the engineering discipline to design, construct, and adapt methods, techniques, and tools for the development of information systems” [5]. The main goal of method engineers is therefore to facilitate the work of software engineers by providing customized methods and tools for software development. To achieve this goal, method engineers perform Method Engineering activities assisted by formalized meta-methods.

A meta-method is a method for defining software development methods [56]. Examples of meta-methods are the Method for Method Configuration (MMC) [4], the assembly-based process proposed by Ralyté *et al.* in [23], and also the methodological approach defined in this thesis. The notions of meta-method (for Method Engineering) and method (for Software Engineering) suggest the existence of some sort of layering or “instance-of” relationship between Method Engineering and Software Engineering. The relationship between these two disciplines is graphically depicted in Figure 2.1.

As Figure 2.1 shows, at the Method Engineering level, method engineers make use of meta-methods and CAME environments to produce (or instantiate) customized

[1] http://www.eclipse.org/modeling/

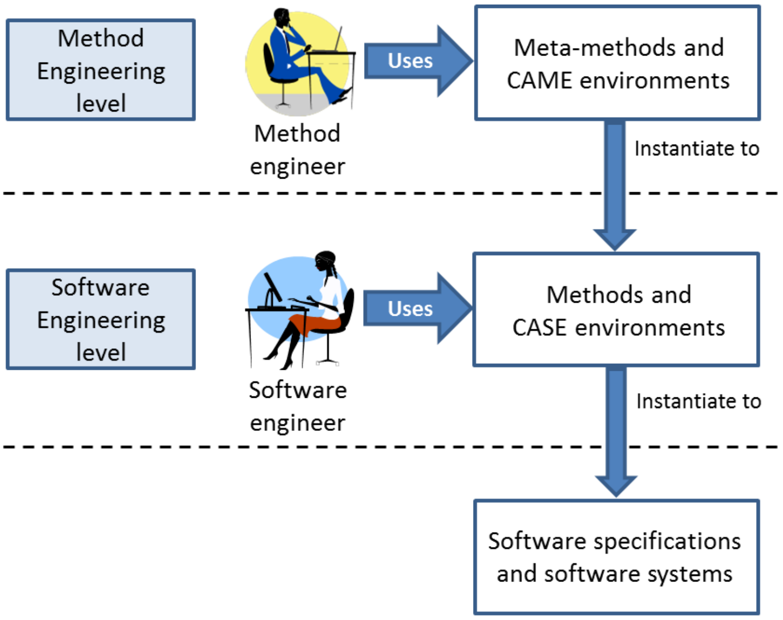

Method Engineering level
Method engineer
Uses
Meta-methods and CAME environments
Instantiate to
Software Engineering level
Software engineer
Uses
Methods and CASE environments
Instantiate to
Software specifications and software systems

Figure 2.1: Method Engineering and Software Engineering

methods and CASE environments for software development. These methods and CASE environments are used by software engineers at the Software Engineering level to produce software system specifications and the final software systems. According to this multi-layer vision, method engineers can be seen as software engineers since they ultimately produce software systems (specifically, software systems for software development). For this reason, method engineers often embody the same group of people as those carrying out Software Engineering. However, in an ideal situation, these two groups will be separate since Method Engineering and Software Engineering involve activities that require different expertise. While method engineers must be proficient in aspects related to software development methods (e.g., maturity appraisal, method specification standards, and method tailoring), software engineers deal with knowledge areas such as software design, configuration management, and software quality [57]. Additionally, software engineers must be knowledgeable about their particular application domain (e.g., avionics software engineers need a great deal of knowledge of aerodynamics).

Despite the differences between Method Engineering and Software Engineering, these two disciplines also have commonalities. One important aspect that the

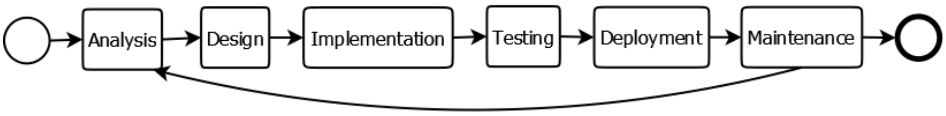


Figure 2.2: Lifecycle of a software system

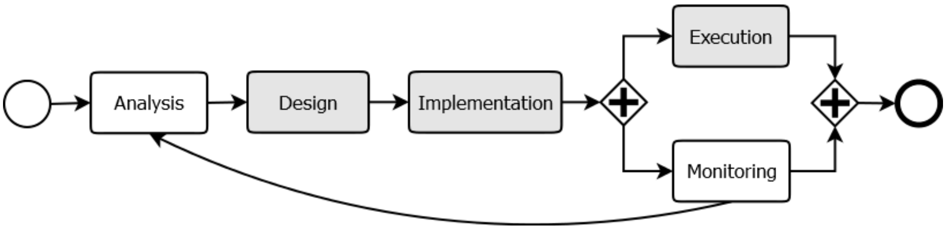


Figure 2.3: Lifecycle of a software development method

two disciplines have in common relates to the phases of their respective lifecycles. On the one hand, the Software Engineering community considers the lifecycle of a software system to be composed of, among others, the following core phases: analysis, design, implementation, testing, deployment, and maintenance [58] (see Figure 2.2). On the other hand, in the Method Engineering literature, we can find lifecycle models of different nature [41, 59, 60]; nonetheless, all these models comprise generic phases that are similar to those proposed by the Software Engineering community. These phases are graphically depicted in Figure 2.3.

As Figure 2.3 shows, the first phase of the Method Engineering lifecycle is the analysis of the method requirements. When the requirements of the method are fully analyzed, the method is designed and implemented. Subsequent executions of the method can be monitored to obtain feedback that is fed to the first phase in order to start another Method Engineering cycle.

Note that the lifecycle of a software system (Figure 2.2) and the lifecycle of a development method (Figure 2.3) differ in the last phases. These differences are mainly due to the fact that, whereas software systems are typically delivered to a customer, development methods are executed in-house (by practitioners of the same company that implements the method). This allows method engineers to perform monitoring activities during the execution of methods in development projects.

In contrast, software engineers test software systems in controlled environments, and, then, the systems are deployed in third-party contexts and maintained based on the feedback provided by users.

In this thesis, we focus on three of the phases that are shown in Figure 2.3. Specifically, we focus on the design, implementation, and execution of software development methods; the other phases are out of the scope of this thesis but they will be addressed in future work (see Section 7.4). The three phases that are covered in this thesis are explained in Sections 2.1.2, 2.1.3, and 2.1.4, respectively. Prior to the description of these phases, a brief note on terminology is given in Section 2.1.1.1.

2.1.1.1 Terminology

In a similar way to most research areas, Software Engineering and Method Engineering present discrepancies in terminology between different authors. One of the most lively debates that is currently active within these fields revolves around the concepts of "method", "methodology", and "process". To differentiate these concepts, in this thesis we adopt the vision that Henderson-Sellers *et al.* share in [3]. According to these authors, the concepts of method, methodology, and process should be interpreted as follows:

Method. A software development method is an approach to perform a software/systems development project, based on a specific way of thinking, consisting, *inter alia*, of guidelines, rules, and heuristics, structured systematically in terms of development activities, with corresponding development work products, and developer roles (played by humans or automated tools) [3]. Cockburn [1] calls this a *Big-M methodology*, which means that a method encompasses absolutely everything that is needed to specify how to develop software: roles, skills, teams, tools, processes, activities, milestones, work products, standards, quality measures, etc.

Methodology. The term methodology can be considered a synonym of method. For the sake of clarity, in this thesis we only use the term methodology when it is part of the name of an existing metamodel, technique, ontology, or tool (e.g., the Agile Methodology Toolset [37] or the Methodology Data

Model [20, 61]). Otherwise, we simply use the term (software development) method.

Process. A process is a way of acting, of doing something. Thus, the way you relocate yourself from home to the work environment follows some predefined – or at least practised and often repeated – process [3]. A process can be seen as a set of actions aimed at accomplishing some result; processes are intangible and may be used in different situations and at different granularity levels. However, to complement a process, there are other things that software engineers must be cognizant of (e.g., the work products to be produced and consumed, and the people and tools that are involved in this production and consumption). As stated above, the overall combination of all the elements that are needed for software development is called method (or methodology). According to this vision, processes can be considered to be part of methods but not vice versa.

The term process is also a source of confusion when it is compared to the term *lifecycle*. In Software Engineering, the term lifecycle is generally used to describe the process that a software system goes through during its life [62]. This process comprises, among others, the phases of analysis, design, implementation, testing, deployment, and maintenance. By analogy, we use the term lifecycle in the context of Method Engineering to denote the phases that a method goes through during its life. In this thesis, we consider the phases of design, implementation, and execution.

2.1.2 Method Design

The design of a software development method involves the conceptual definition of all the elements that comprise the method, such as the participants involved, the work products to be developed, and the sequence of tasks. Up to now, software development methods have frequently been described purely in a textual manner. However, more recently, various authors have defined a number of method specification languages (see e.g. [63, 64, 65, 66]). Additionally, several standardization efforts have been performed. These efforts can be seen in the ISO/IEC 24744 [67, 68] and SPEM 2.0 [18] standard initiatives. Another standards from the field of Business Process Management can also be mentioned. An example of these standards is BPMN 2.0 [49]. Nonetheless, these standards are more oriented towards the specification of the process-related aspects of Method Engineering.

Below, Section 2.1.2.1 provides an overview of SPEM 2.0 since this standard is the main method specification language that is used in this thesis. We selected SPEM 2.0 because it is the most widely acknowledged standard in both academic circles and software industry [69]. However, this language has some process support limitations, which we resolve using BPMN 2.0. For this reason, we introduce the BPMN 2.0 standard in Section 2.1.2.2.

2.1.2.1 The SPEM 2.0 Standard

The Software and Systems Process Engineering Metamodel (SPEM) [18] is a standard language that was initially published by the Object Management Group (OMG) in 2002. Six years later, in 2008, the OMG released the second version, which aimed to fix defects that were found in the first version of the standard. Unlike the first version, the second version of SPEM is widely accepted by practitioners; they use SPEM 2.0 for developing methods in different domains, especially multi-agent systems, software product lines, and embedded real-time systems [69].

The OMG defines SPEM 2.0 as a "process engineering metamodel as well as a conceptual framework, which can provide the necessary concepts for modeling, documenting, presenting, managing, interchanging, and enacting development methods and processes". The scope of SPEM 2.0 is limited to the minimal set of elements necessary to define methods that are independent of parameters such as development paradigm (e.g., agile, code-centric, or model-driven), degree of formalization, and cultural backgrounds. Nonetheless, the primary focus of SPEM is software development projects.

The core of **the SPEM 2.0 metamodel comprises three basic elements: task, role, and work product**. The relationships between these elements are illustrated in Figure 2.4. Specifically, tasks – which represent basic units of work (e.g., "specify business logic") – are performed by roles. The roles of the method – which define sets of related skills, competencies, and responsibilities (e.g., "analyst") – are used by tasks to define who performs them as well as to define a set of work products that the roles are responsible for. These work products – which represent artifacts that are consumed, produced, or modified during the

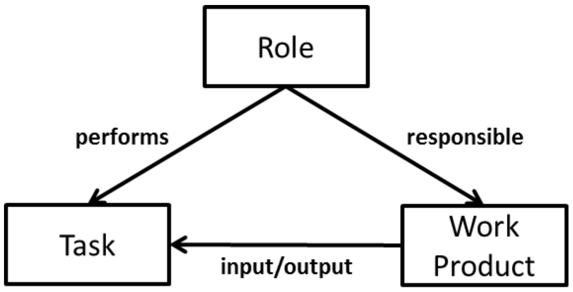


FIGURE 2.4: Core elements of SPEM 2.0

method (e.g., "business logic model") – are in turn related to the tasks to specify the inputs and outputs of these tasks.

One important characteristic of SPEM 2.0 is that **it separates method content and development processes**. The content of a method defined in SPEM 2.0 provides generic and reusable definitions of tasks, roles, and work products; these definitions are independent of any development context. On the other hand, development processes take content elements and relate them into partially-ordered sequences that are customized to specific types of projects. To illustrate this idea, let us consider a software development project that develops an application from scratch. In this project, software engineers will perform development tasks in a similar way to a project that extends an existing software system; however, these tasks will be performed at different points in time and with different emphasis (e.g., software engineers will perform the steps of these tasks differently, or assume different inputs).

Figure 2.5 provides an overview of how the key concepts of SPEM 2.0 are positioned to represent either method content or development process. Method content is primarily expressed using work product definitions, role definitions, task definitions, and guidance; categories (e.g., domains, disciplines, and role sets) can also be defined to categorize method content as well as to define tree-structures of nested categories (allowing the user to navigate and browse method content based on these categories). The elements of type guidance (e.g., whitepapers, checklists, and examples) are defined in the intersection of method content and process because they can provide information with respect to the elements of both sets. On the other hand, processes are defined in SPEM 2.0 by using activities.

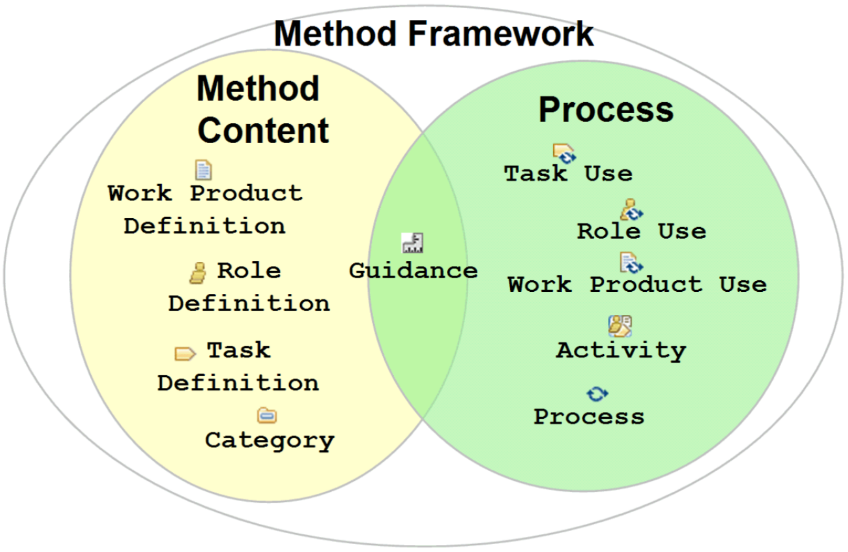


Figure 2.5: Method content versus process [18]

The elements of type activity can be nested to define breakdown structures as well as related to each other to define a flow of work (i.e., a process); activities also contain task uses, role uses, and work product uses, which represent references to method content elements.

Another important characteristic of SPEM 2.0 is that **it enables the definition of process patterns for rapid process assembly**. Process patterns are reusable building blocks for creating new development processes. Selecting and applying process patterns can be done in two different ways: “copy and modify” and “activity use”. The former allows method engineers to individually modify the pattern’s content when the pattern is applied in a process (since the pattern’s content is copied into the process, not referenced). The latter represents a way of reusing commonly recurring activities. To reuse an activity, it must be factored out into a pattern so that the pattern can be applied repeatedly in a process by simply creating references to the pattern. Unlike the “copy and modify” operation, all changes in a pattern are automatically reflected in all the processes that apply the pattern.

2.1.2.2 The BPMN 2.0 Standard

The Business Process Model and Notation (BPMN) [49] is a process modeling language that was initially developed by the Business Process Management Initiative (BPMI), a consortium that consisted mainly of software companies. The first version of BPMN was published in 2004 and officially accepted as an OMG standard in 2006. Five years later, in 2011, the OMG released the second version of the standard, which aimed to fix defects that were found in the first version. Some changes were the adoption of XML and the formalization of the execution semantics.

An important characteristic of BPMN 2.0 is that it provides an intuitive graphical notation that can be readily understandable by all process stakeholders. In this section, we provide an overview of the most relevant building blocks of this notation. These elements are organized in five categories: activities, events, sequence flows, gateways, and swimlanes.

Activities. An activity is a generic term for work performed within a process. Activities can be atomic (tasks) or decomposable (sub-processes and call activities); also, activities are executed by either a system (automatic) or humans. The tasks that are automatic can be either service tasks or script tasks; the tasks that are executed by humans can be user tasks (if they require the use of a software application) or manual tasks (if they must be performed without any software support). The graphical object that represents activities is a rounded rectangle, as Figure 2.6 shows. The different types of tasks are specified using markers that are positioned in the upper left corner of the rectangle.

Events. An event is something that "happens" during the course of a process. They are used to start or end a process, and also to manage specific actions during a workflow (e.g., receiving and sending messages between participants). Events are depicted as circles with open centre to allow internal markers to differentiate various types of events. There are three main types of events, based on when they affect the workflow: start, intermediate, and end. Figure 2.7 shows the complete set of events that are defined by BPMN 2.0.

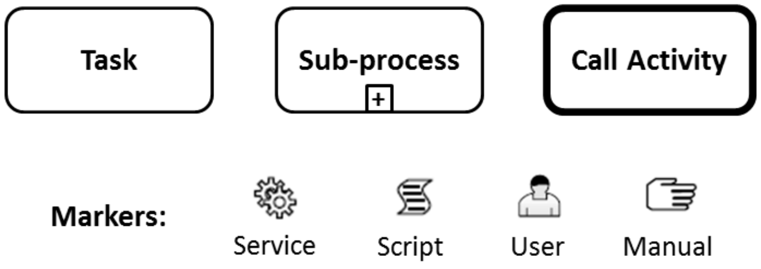


Figure 2.6: Activity types defined by BPMN 2.0

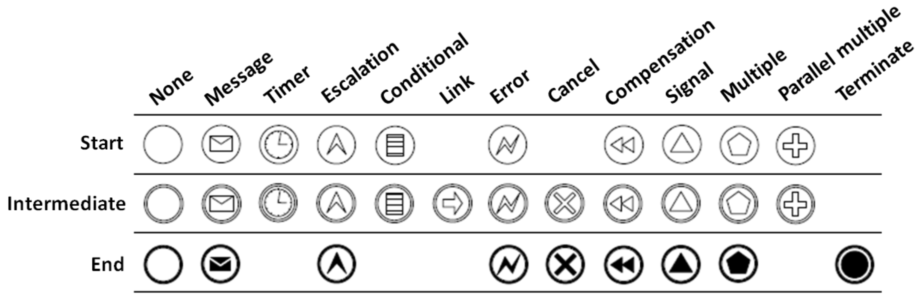


Figure 2.7: Event types defined by BPMN 2.0

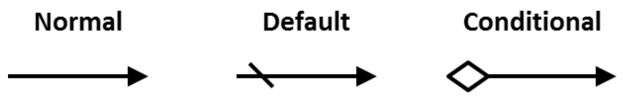


Figure 2.8: Sequence flow types defined by BPMN 2.0

Sequence flows. Sequence flows are used to connect the activities that are defined within a process. The connection between two activities establishes the order in which the activities are executed. There are three types of sequence flows: normal, default, and conditional. The graphical object that represents sequence flows is a line with a solid arrowhead (as Figure 2.8 graphically illustrates).

Gateways. Gateways are used to control how the process progresses through the sequence flows as these flows converge and diverge. If the flow does not need to be controlled, then a gateway is not needed. The term "gateway" implies

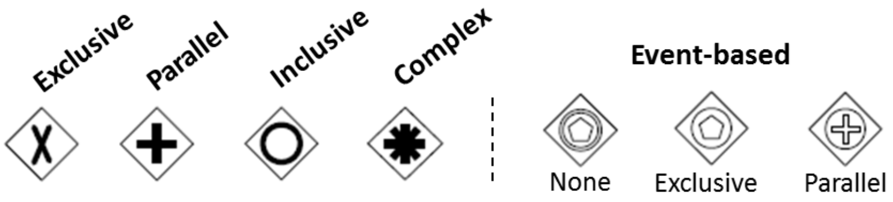


FIGURE 2.9: Gateway types defined by BPMN 2.0

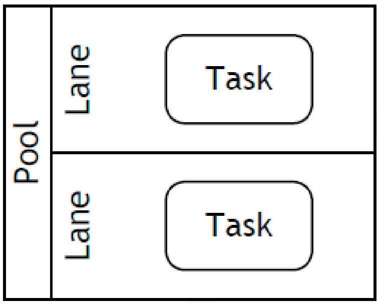


FIGURE 2.10: Swimlane types defined by BPMN 2.0

that there is a gating mechanism that either allows or disallows passage through the gateway. Internal markers indicate the type of behavior control. Figure 2.9 depicts the different types of gateways provided by the BPMN 2.0 notation.

Swimlanes. Swimlanes enable the grouping of activities based on a particular criterion. There are two types of swimlanes: pools and lanes (see Figure 2.10). Pools (also called participants) represent responsibilities for activities in a process. A pool can be, for instance, an organization, a role, or a system. Lanes subdivide pools or other lanes hierarchically.

Summary and Discussion

The use of SPEM 2.0 and BPMN 2.0 brings important benefits to the work that has been developed in this thesis. Some of these benefits are summarized in the analysis of SPEM 2.0 and BPMN 2.0 that is presented in Appendix A. The most relevant benefits are the following:

- *Standardization:* the use of SPEM 2.0 and BPMN 2.0 allowed us to define a Method Engineering approach (as well as to implement a supporting CAME environment) that allows method engineers to carry out the design of methods using standardized concepts. This is in line with one of the research initiatives that was foreseen by Henderson-Sellers *et al.* in a state-of-the-art review that was elaborated in 2010 [3]. In this review, the authors stated that one of the likely topics for research initiatives over the next few years would be the creation of a new generation of CAME environments built upon internationally standardized metamodels.

- *Suitable primitives for method modeling*: by combining the SPEM 2.0 and BPMN 2.0 standards, our Method Engineering approach enables the specification of the product and process parts of methods, including also the method producers and the method tools (see Appendix A).

- *High reusability*: the SPEM 2.0 standard promotes reusability by means of the separation of method content and processes. This separation allows method engineers to design and manage libraries/repositories of reusable method content. Thus, SPEM 2.0 can be used as a framework for the construction of software development knowledge bases, where reusable method content can be stored in a standardized manner.

2.1.3 Method Implementation

The implementation of a software development method involves the construction of an integrated software environment that contains all the tools necessary to support the method. The emergence of the first environments of this kind dates back to the early seventies, when Teichroew *et al.* introduced PSL/PSA [70, 71]. One decade later, in 1982, the term "Computer-Aided Software Engineering" (CASE) was coined by the software company *Nastec Corporation*, which developed an integrated graphical and textual editor called "GraphiText" [72]. After that, software companies began to borrow ideas from hardware manufacturing in order to apply them for software development. The ultimate goal was to obtain CASE environments that allowed them to develop software at lower cost while maintaining high quality and meeting customer demands.

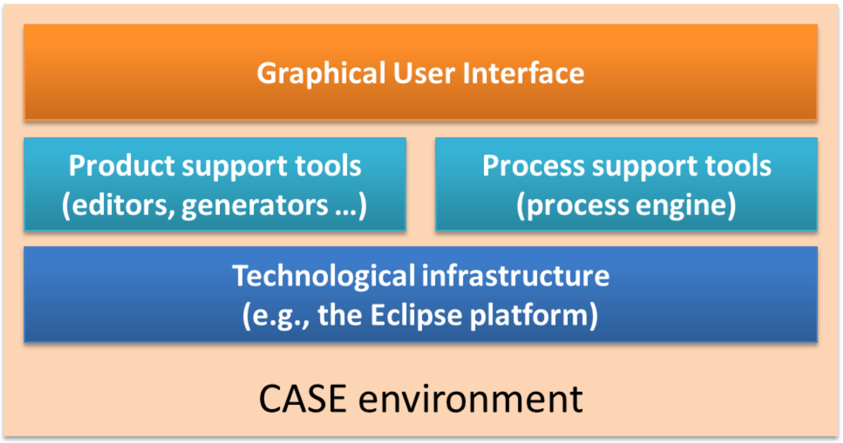

Graphical User Interface
Product support tools (editors, generators ...)
Process support tools (process engine)
Technological infrastructure (e.g., the Eclipse platform)
CASE environment

FIGURE 2.11: General architecture of a CASE environment

2.1.3.1 Computer-Aided Software Engineering

A CASE environment is a macro-system that, in its ideal form, provides software support for all aspects of software engineering. Rather than being a loosely coupled collection of tools, CASE environments are designed to optimize the complimentary benefits of different tool types (from simple textual editors to more sophisticated tools such as graphical editors, code generators, and process engines) [73].

Figure 2.11 graphically depicts the general architecture of a CASE environment. To provide complete method support, CASE environments must meet two major requirements. First, they must incorporate a process engine to support the execution of the method process part. Process engines provide a set of enactment facilities (e.g., task orchestration, task automation, and constraint enforcement) that guide software engineers throughout the development process and also partially automate the process performance. Second, CASE environments must also support the management of the products that are consumed/produced during the process execution. To this end, the process engine must be able to invoke the software tools that enable the creation and manipulation of the method products.

Due to the potential of CASE technology, many scholars claimed that CASE environments would substantially reduce the costs of developing software, standardize system specifications, and improve the quality of information systems [74, 75]. However, evidence from actual use of CASE environments showed a rather different picture [76]: software development projects continued to run over-time and over-budget, and software products were of low quality and inefficient. The failure of CASE technology was due to several reasons. First, CASE environments failed to automate important software development tasks, such as those related to project management. Second, adopters of CASE technology underestimated the training costs. Third, CASE environments offered very limited flexibility; that is, they contained fixed hard-coded tools that lacked the capability to be adapted to context needs.

2.1.3.2 MetaCASE Environments

The inflexibility of traditional CASE tools led to the emergence of new technology: metaCASE environments [76, 77, 78] (also known as customizable CASE environments [20, 79] or CASE shells [80, 81]). The main goal of metaCASE environments was to provide metatools that made the construction and adaptation of CASE environments quicker and easier. In general, metaCASE environments work on the philosophy that all CASE environments have common standard characteristics (such as the need for a technological infrastructure and a graphical user interface), but individual CASE environments differ in the functionality that they provide (i.e., in the method that they support). Therefore, metaCASE environments provide mechanisms to configure this functionality while maintaining common standard characteristics.

Figure 2.12 illustrates the most frequent approach to CASE environment construction using metaCASE technology. This approach is called “specify and generate” [40, 42, 82, 83]. In particular, the metaCASE environment provides a common framework (i.e., a common graphical user interface and technological infrastructure) for all the CASE environments that can be generated using the metaCASE system. In order to leverage this common framework, the user must specify the details of the software tools that meet his/her situational needs. To this end, the user can employ a description language that is provided by the metaCASE environment. The specification of the software tools is then “plugged” into the common framework so that the final CASE environment can be automatically generated.

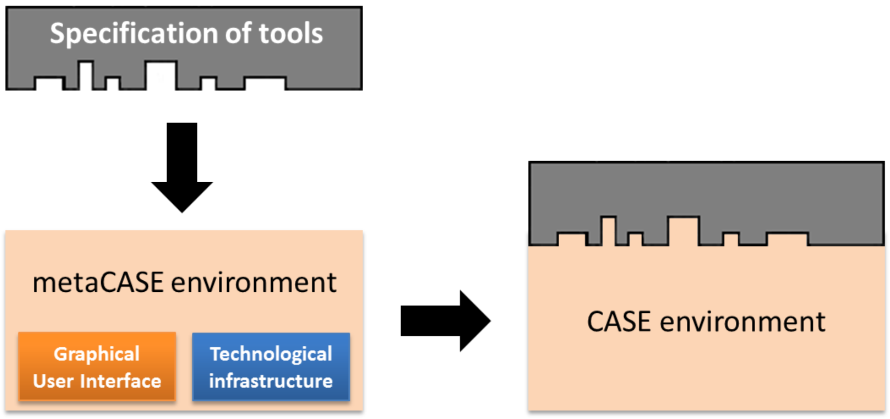


FIGURE 2.12: CASE construction using metaCASE (adapted from [83])

Summary and Discussion

One of the major benefits of metaCASE systems is the drastic reduction of the cost (in terms of time, people, and money) that is required to develop CASE environments, since these environments can be developed without writing a single line of code.

In this thesis, we do not directly use metaCASE technology; nonetheless, we apply some of its principles in order to support the method implementation phase of the Method Engineering lifecycle. Similarly to the CASE environments that are obtained using metaCASE systems, the software environments that are obtained using our CAME environment are divided into two main parts. One of these parts is static; that is, it comprises components that are included in all the software environments that are generated using our CAME environment (regardless of the method that is defined by method engineers). The other part is dynamic; that is, it comprises components that depend on the specified method. During the generation of a software environment, our CAME environment "plugs" these dynamic components into the static part (which corresponds to the common framework of metaCASE systems). The dynamic components are tools such as textual/graphical editors, which are built using metatools that are provided by our CAME environment. These tools are encapsulated in reusable assets and associated to conceptual method elements. Thus, we establish the linkage between the conceptual and technical aspects of methods.

2.1.4 Method Execution

The method execution involves the enactment of method instances in specific software development projects. This enactment is tool-assisted; that is, the integrated environment that is obtained in the implementation phase must provide guidance for software engineers during the entire process of software development. Up to now, most CASE environments supporting development methods contain the method logic scattered through the system. This results in monolithic applications that are difficult to evolve and expensive to maintain. To resolve this problem, an executable definition of the method is needed. With an executable definition, the method knowledge is centralized and the method can be modified easily, resulting in the immediate update of the corresponding CASE environment.

Recently, various languages have been proposed to enable the definition of executable methods and processes. Some examples of these languages are the XML Process Defition Language (XPDL) [84], Yet Another Workflow Language (YAWL) [85], Web Service Business Process Execution Language (WS-BPEL) [86], and the Business Process Model and Notation (BPMN) [49, 87]. In subsection 2.1.4.1, we describe the operational aspects of BPMN 2.0 since this standard is the language that is used in this thesis to overcome the lack of executability of SPEM 2.0 [88, 89]. An extensive review of process modeling languages can be found in [90].

2.1.4.1 Operational Aspects of BPMN 2.0

The execution semantics of BPMN have been fully formalized in the second version of the specification [49]; thus, BPMN 2.0 offers the possibility to define models that can be executed in process engines that support the operational semantics of the standard. Thanks to this feature of BPMN 2.0 (and also to its intuitive graphical notation), both the business and technical sides of an organization can share a common language that meets their respective needs for ease of use and executability. This is unlike other languages, such as WS-BPEL, which are generally optimized for the operation and interoperation of Business Process Management systems, but this capability renders them less suited for direct use by humans to design, manage, and monitor business processes. Despite this limitation, BPMN

2.0 provides a mapping to WS-BPEL so that BPMN 2.0 process models can also be executed by process engines that support this language.

The execution semantics of BPMN 2.0 describe a clear and precise understanding of the operation of some of the elements that are proposed by the standard. The elements that do not specify the details needed to execute them are called non-operational. An example of non-operational element is *Manual Task* [49]. For the BPMN 2.0 elements that are operational, the standard describes their execution semantics informally. Specifically, for each element, a textual description of its execution semantics is given first; then, where relevant, this textual description is followed by a list of exception issues and supported workflow patterns.

Summary and Discussion

Among the many benefits of BPMN 2.0 (e.g., support not only for simple processes but also for collaborations and choreographies; a detailed mapping from BPMN 2.0 to WS-BPEL; and an interchange format that can be used to exchange BPMN 2.0 definitions between tools of different vendors), one of them is particularly noteworthy: BPMN 2.0 is the first standard that combines (within the same process model) a user-friendly graphical notation and the technical details of an executable model. In this thesis, we leverage this major capability of BPMN 2.0 to overcome the limitation of SPEM 2.0 with respect to process executability (see Appendix A).

2.2 Model-Driven Engineering

In the Software Engineering literature, several terms have been proposed to describe approaches that foster the use of models as primary artifacts for software development (see Figure 2.13). The first of these terms was introduced in 2001, when the OMG launched the Model-Driven Architecture (MDA) [91]. MDA is a framework that makes intensive use of OMG's standards to support the creation of machine-processable and technology-independent models; once these models are defined, the MDA enables their automatic transformation into platform-dependent code [92]. One year later, in 2002, Kent [93] proposed the broader term Model-Driven Engineering (MDE). This term is widely used in the literature for referring to a more general approach than the one proposed by MDA

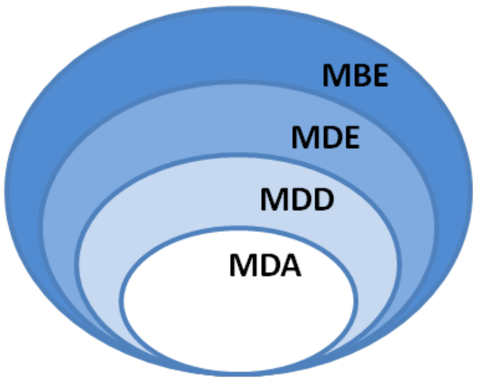


Figure 2.13: Relationships between model-driven acronyms [98]

[94, 95, 96]: MDE is not restricted to the standards of the OMG. Another term that has been used is Model-Driven Development (MDD). This term is narrower than MDE since MDD focuses on the use of models for the generation of system implementations, while MDE encompasses other uses, such as model-driven reverse engineering and model-driven evolution [97]. Finally, the term Model-Based Engineering (MBE) has also been used as a softer version of MDE. In contrast to MDE, in MBE models do not drive the process of software development [98].

Due to the nature of the Method Engineering approach that is presented in this thesis (i.e., it is neither restricted to the standards of the OMG nor to code generation, but it heavily relies on models), we use the term MDE.

2.2.1 Defining Model-Driven Engineering

From the inception of computer science, researchers have strived to raise the abstraction level at which software engineers write programs. The first compiler was a major achievement because, for the first time, it let software engineers specify *what* the system should do rather than *how* it should do it. Since then, researchers have always attempted to raise the level of abstraction even more. MDE is the result of this effort. Instead of requiring software engineers to code every detail of the system (using a programming language), MDE allows them to model the functionality of the system and also its overall architecture using high-level primitives. The resulting models can then be used to automate many of the complex and routine tasks that are involved in the process of software development (e.g., providing support for the system's persistence, interoperability, and distribution) [99].

MDE is therefore a development paradigm where models represent first-class citizens for the construction of software systems. Some of the potential benefits of MDE are the following: simplification of the design process, improvement in software quality, increase in productivity, shorter development time, and enhanced communication between the individuals and teams working on the system [43].

Despite the potential benefits of MDE, some scholars claim that the industrial application of MDE is currently minimal [97]. However, a recent study that surveyed 450 MDE practitioners and performed in-depth interviews with 22 more suggests otherwise: MDE is more widespread than commonly believed. The study illustrates that MDE is used in many different ways, ranging from industry-wide efforts to define models for an entire application domain to restricted uses of MDE in the generation of code for a single application in a single company [97].

Of all the successful techniques applied in MDE, three are the most relevant for this thesis since they lay the foundations of the overall thesis work. These techniques are metamodeling, model transformations, and models at runtime. We explain these three techniques in Sections 2.2.2, 2.2.3, and 2.2.4, respectively.

2.2.2 Metamodeling

A model is a coherent set of formal elements describing something (e.g., the architecture of a system or the schema of a database) at a high level of abstraction and is built for some purpose (e.g., completeness checking or code generation) [100]. Models are expressed in a modeling language whose syntax must be defined somehow. Since modeling is an appropriate technique to formalize knowledge, we can define this syntax by building a model of the modeling language – a so-called metamodel.

Metamodeling plays a key role in MDE. Atkinson *et al.* investigate in [99] the technical foundations of MDE and discuss the role of metamodeling in a supporting infrastructure. To define metamodels, metamodeling languages such as Ecore can be used. These languages let us define metamodels using object-oriented constructs such as classes, associations, and generalization [101]. With the potential of metamodeling languages, software engineers can easily define modeling languages that are adapted to their specific application domain. This type of language is called Domain-Specific Language (DSL).

2.2.2.1 Domain-Specific Languages

A DSL is a language (e.g., a modeling language or a programming language) that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain [102].

DSLs are not a new topic, but the current emphasis on MDE has focused the interest of both academia and industry on this kind of languages. Examples of DSLs abound, including well-known and widely-used languages such as LaTeX, SQL, HTML, SPEM, and BPMN. We can also consider as DSLs old programming languages such as Cobol, Fortran, and Lisp since they came into existence as dedicated languages for solving problems in a certain area (in this case, business processing, numeric computation, and symbolic processing, respectively) [102].

Summary and Discussion

In order to support the conceptual definition of methods, in this thesis we define a DSL that integrates concepts from two standard DSLs: SPEM 2.0 and BPMN 2.0. Defining and using a DSL always involves both risks and opportunities. In our proposal, it brings two general benefits:

- DSLs allow methods to be expressed in the idiom and at the level of abstraction of the software development domain; consequently, domain experts themselves can understand, validate, modify, and even develop methods using the DSLs.
- Methods that are developed using DSLs are concise, self-documenting to a large extent, and can be reused for different purposes.

In addition of these general benefits, we can mention other benefits that relate to the fact that DSLs allow methods to be represented as machine-processable models. These benefits are the following:

- Method models enable the automatic generation of documentation of the methods in different formats, such as HTML or plain text.
- Methods become easier to maintain and easier to navigate (compared to methods that are formalized in textual documents).
- A method model facilitates the communication between the people that is involved in a project (who typically have different levels of expertise and different roles) since modeling helps with getting a better overview of the method by providing higher levels of abstraction.
- CASE environments can execute method models at runtime to assist software engineers during the course of the projects.

In contrast to all of the above benefits, some drawbacks can also be emphasized. These drawbacks mostly relate to the costs of designing, implementing, and learning the DSL.

2.2.3 Model Transformations

In the context of MDE, models are the main development artifacts and model transformations are among the most important operations applied to models [103]. A model transformation can be defined as a set of rules that together describe how a model in a source language can be transformed into a model in a target language. A single transformation rule is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language [92].

The practical application of MDE may imply the use of several kinds of model transformations. In the following, we present a list of four criteria for classifying model transformations; these criteria have been extracted from the taxonomy presented by Mens *et al.* in [104]. Based on this taxonomy, model transformations can be classified according to:

Source and target languages. According to this criterion, a distinction can be made between *endogenous* and *exogenous* transformations. The former are transformations between models that are expressed in the same language;

the latter are transformations between models that are expressed using different languages. Refactoring is an example of endogenous transformation, while reverse engineering is an example of exogenous transformation.

Abstraction level. A *horizontal* transformation is a transformation where the source and target models reside at the same abstraction level. Typical examples are refactoring and migration. A *vertical* transformation is a transformation where the source and target models reside at different abstraction levels. A typical example is refinement, where a specification is gradually refined into a full-fledged implementation by means of successive refinement steps that add more concrete details.

Source and target models. Based on this criterion, a distinction can be made between *model-to-text* (M2T) and *model-to-model* (M2M) transformations. M2T transformations take a model as input and produce plain text as output (e.g., source code, documentation, or configuration files). On the other hand, M2M transformations take a model as input and produce another model as output. Code generation is an example of M2T transformation, while model refinement is an example of M2M transformation.

Model semantics. A final distinction can be made between model transformations that merely transform syntax, and more sophisticated transformations that also take the semantics of the models into account. As an example of *syntactical* transformation, consider a parser that transforms the concrete syntax of a model into an abstract syntax. The abstract syntax is then used as the internal representation of the model on which more complex *semantic* transformations (e.g., optimisation) can be applied.

In addition to the above transformation kinds, we can also find in the literature different types of approaches for implementing model transformations [104, 105, 106, 107]. These approaches can be classified as follows:

Direct model manipulation. Following this approach, models are accessed by means of a general-purpose language (such as Java, Python, or C#), generally using an API that has been specifically designed to manipulate machine-processable models. The main advantage of this approach is that developers need little or no extra training to implement the transformations. An important disadvantage is that the API usually restricts the transformations that can be implemented. Additionally, since the language is general-purpose and not domain-specific, it lacks suitable abstractions for specifying model transformations.

XSLT. In this approach, models are serialized to XML. Then, model transformations are implemented using the XSLT language, which is a standard technology for transforming XML. This approach is discouraged due to severe scalability limitations: XSLT transformations quickly become non-maintainable implementations because of the verbosity and poor readability of XSLT [107].

Graphs. Following this approach, input models are represented as attributed typed graphs and model transformations are specified as sets of graph rewriting rules. A rule consists of a graph to match – commonly referred to as left-hand side (LHS) graph – and a replacement graph – commonly referred to as right-hand side (RHS) graph. If a match is found for the LHS graph, then the rule is triggered. The execution of the rule causes the RHS graph to replace the matched sub-graph.

Model transformation languages. In this approach, a DSL for implementing model transformations is used. These languages provide constructs for e.g. specifying source and target metamodels, navigating models, and implementing transformation mappings. Some examples of DSLs for model transformations are ATL [103], RubyTL [101], and MOFScript [108]. These languages can in turn be classified as *declarative* or *operational*. Declarative languages focus on the *what* aspect (i.e., they focus on what needs to be transformed into what by defining a relation between the source and target models). Operational approaches focus on the *how* aspect (i.e., they focus on how the transformation itself needs to be performed by specifying the steps that are required to derive the target models from the source models) [104].

Templates. This approach only applies to M2T transformations. Specifically, a template engine (e.g., Velocity[2]) is used to generate text files taking as input data-like representations of models.

[2] http://velocity.apache.org/engine/devel/

Summary and Discussion

In this thesis, we take full advantage of the technology that is provided by model transformations, since it allows us to support the method implementation phase of the Method Engineering lifecycle. By means of model transformations, we leverage method models using them for the construction of the supporting software environments. Thanks to the use of model transformation technology, we reap the following benefits:

- *Bridging conceptual and technical spaces*: by means of M2T transformations, we bridge the gap between the conceptual design of methods and the technical details of their implementations. This is equivalent to one of the most common uses of M2T transformations in Software Engineering: the automatic generation of source code (which resides at the technical space) from high-level models (which reside at the conceptual space).
- *Automation*: the CASE environment generation process is automatic in our approach. This is a major benefit since obtaining a tool automatically, in general, increases the quality of this tool [43]. Because the tool is specified in high-level models, the quality of the tool depends on the generator (i.e., the M2T transformation). Therefore, the quality can greatly increase because we can let our best people work on the M2T transformation. Furthermore, all of the best practices that we eventually learn can be included in the transformation, and, thus, they will be automatically applied in all of the tools that are subsequently created.
- *Model merging*: by means of M2M transformations, we allow method models to be automatically extended with reusable method parts. These parts are instances of one or more concepts of the DSL for method design that we define in this thesis; therefore, all of the models that are involved in the M2M transformations (i.e., the input and output models) are instances of the same metamodel.

2.2.4 Models at Runtime

A prominent approach to extend the applicability of MDE is to bring development models to the runtime environment [109]. In this way, the modeling effort made at design time is not only useful for producing the system but it can also drive the system's behavior during execution [110]. This approach is referred to as *models at runtime* (sometimes called *models@run.time*).

A model at runtime can be defined as a causally connected self-representation of the associated system that emphasizes the structure, behavior, or goals of the system from a problem space perspective [109]. One of the major benefits of models at runtime is that they enable controlled ongoing design [109]; that is, they allow software engineers to incorporate new design decisions into a running system by simply updating the models.

For the sake of comprehension, it is important to describe the differences among *development* models, *executable* models, and *runtime* models[3]. When the models are produced in a MDE process, they are named development models. These models are in an abstraction level above the code level [111]. Examples of development models are use cases and architectural models. If development models are expressive enough to be automatically executed, they are considered as executable models. Executable models can be executed by translating them into executable code [112] or by using an interpreter/engine that directly executes the model [113]. When the models are used at runtime, they are considered runtime models. In contrast to development models, runtime models are used to reason about the operating environment and runtime behavior for some purpose (e.g., determining an appropriate form of adaptation) [114].

Another source of confusion is the contrast of models at runtime with historic efforts in the field of reflection. In [109], the authors state the similarities and differences between these two concepts: *reflection* and *models at runtime*. These concepts are similar in the sense that both are concerned with defining representations of the underlying system that are causally connected. Causal connection means that if the system changes, the representations of the system (i.e., the models) should also change, and vice versa. On the other hand, runtime models and reflection differ in that reflection seeks models that are intrinsically related to the computation model; hence, these models are based on the solution space and tend to be rather low-level. In contrast, runtime models reside at a much higher level of abstraction since they are based on the problem space. In summary, the idea

[3]In this thesis, we consider the term "runtime model" as a synonym of "model at runtime".

of models at runtime is built on reflection but seeks to move from the solution space up to the problem space.

Summary and Discussion

In this thesis, we bring method models to the runtime environment in order to support the method execution phase of the Method Engineering lifecycle. By using models at runtime, we leverage the modeling effort that is made at design time to provide assistance for software engineers during the course of development projects.

In addition to the above benefit, models at runtime also play a significant role in **reducing the internal complexity** of software systems. Note that, in Software Engineering, when models at runtime are used, the component that executes the models (i.e., the model interpreter) replaces a large amount of the source code of the final software system; for instance, it replaces the code that implements the concepts, attributes, and relationships that are contained in the models. This code, when models at runtime are not used, is typically scattered throughout the system and has to be generated (or at least partially generated) from the models. This increases the internal complexity of the system, and, in turn, the complexity of the M2T transformations that are in charge of its generation. In the context of our Method Engineering approach, the use of method models at runtime simplifies the internal structure of the CASE environments: we only have to include a model interpreter, rather than scattering through the system all the information that is contained in the method models. Thus, the complexity of the M2T transformation that is in charge of the CASE environment generation is reduced to a large extent.

Models at runtime, besides reducing internal complexity, also facilitate model changes. This is because these changes can be performed without stopping the running software system and also without requiring an explicit system regeneration. In the context of Method Engineering, this capability of models at runtime can contribute to turn **method evolution** into reality. Changes in project characteristics, which cause unexpected changes in the method models, can be automatically translated into changes in the supporting software environments (without having to stop and regenerate these tools), thereby keeping methods and tools always synchronized. Providing support for method evolution is outside of the scope of this thesis.

2.3 Eclipse-based Technologies

Software engineers need a variety of tools to properly perform the full development process; additionally, these tools must be integrated and work well together. For this reason, the community of software engineers must develop tools in ways that increase the likelihood of their interoperation with other tools. In 2001, IBM addressed this issue by providing a common platform that facilitated the integration of diverse software products [115]. This platform was called Eclipse.

The term "Eclipse" was a wordplay of IBM, who wanted to "eclipse" Microsoft Visual Studio, the primary competition at the time. After more than a decade of existence, Eclipse has had a lasting influence, which is reflected in its high commercial acceptance as well as in its many contributions to different Software Engineering areas, such as MDE, ambient intelligence, process engineering, and software testing.

2.3.1 The Eclipse Platform

In words of the Eclipse foundation[4], Eclipse is a community for individuals and organizations who wish to collaborate on commercially-friendly open source software. Its projects are focused on building an open development platform comprised of extensible frameworks, tools, and runtimes for building, deploying, and managing software across the lifecycle.

Figure 2.14 shows the general architecture of the Eclipse development platform. At the heart of Eclipse is a mechanism for dynamically discovering, loading, and running software components (which, in the context of Eclipse, are called plug-ins). An Eclipse plug-in is the smallest unit of functionality that can be developed and delivered separately; thus, plug-ins determine the platform functionality and whether it operates as an Integrated Development Environment (IDE) or as a general-purpose application [116]. General-purpose applications that are built upon the Eclipse platform are collectively known as Rich Client Platform (RCP) products.

[4]http://www.eclipse.org/

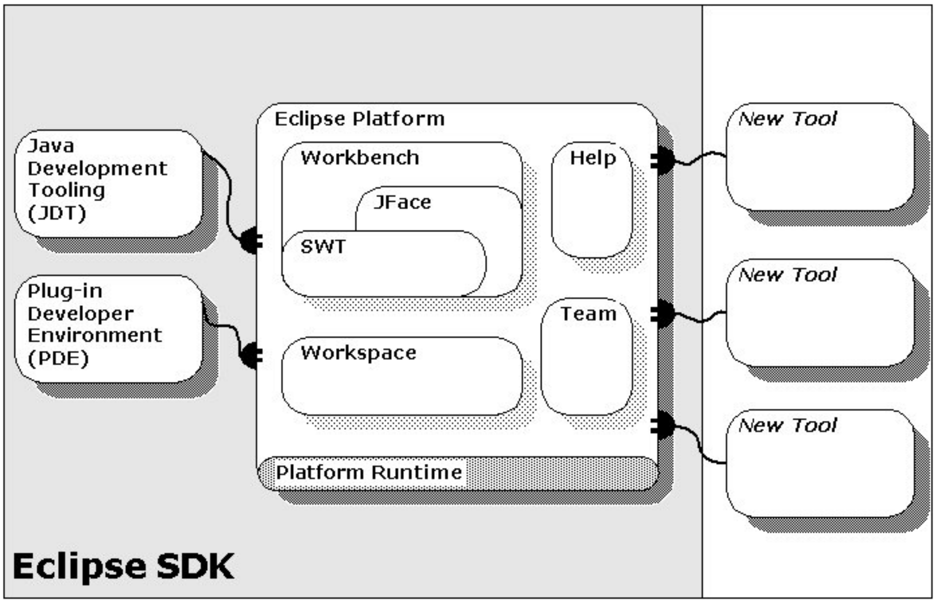


FIGURE 2.14: Eclipse architecture (from http://www.eclipse.org/)

In addition to the plug-ins that can be loaded and run in Eclipse, the platform itself also holds much built-in functionality. The main constituents of the Eclipse platform are the following:

Platform runtime. The platform runtime dynamically discovers plug-ins and maintains information about the plug-ins and their services in a platform registry. Plug-ins are loaded and launched when required, according to the user's actions [116]. The platform runtime is implemented using the OSGi framework.

Workspace. The Eclipse workspace allows the user to manage resources (i.e., projects, files, and folders) that are stored in the file system.

Workbench and User Interface toolkits. Eclipse is built upon a workbench that provides the overall structure of the platform and presents an extensible User Interface to the user. The workbench is built from two toolkits: the Standard Widget Toolkit (SWT) and JFace. The former is a widget set and graphics library that integrates with the native window system. The latter is a toolkit that is implemented using SWT and simplifies common User Interface programming tasks.

Help system. The help system provides the user with building blocks to structure and contribute documentation to the platform.

Team support. The Team component provides repository tooling integration into Eclipse and a universal compare facility. The compare facility implements compare/merge components; differencing engines; integration and creation of patch files; and comparison and merging with the local history.

Other utilities. Other utility plug-ins supply functions such as searching resources. Two of the most important utilities that are provided as part of the Eclipse Software Development Kit (SDK) are the Java Development Tools (JDT) and the Plug-in Development Environment (PDE). While JDT implements an IDE supporting the development of any Java application, the PDE allows the user to build and deploy Eclipse plug-ins and RCP products. One noteworthy aspect of the PDE is the functionality that is provided by the *product configuration files*. A product configuration file is an XML document that defines the characteristics of an Eclipse RCP product (e.g., the icons, splash screen, and plug-ins/features[5] that comprise the product). This XML document can be used by the PDE to automatically generate a fully functional RCP product (which only incorporates the elements that are specified in the product configuration file). The file extension of the product configuration files is *.product*.

In addition to all of the above functionality and capabilities of Eclipse, it is also important to highlight that, behind Eclipse, there is an extensive community of users, researchers, and developers. This community has created a wide variety of tools and frameworks for the Eclipse marketplace[6], which nowadays contains more than a thousand tools, including business process management tools, modeling frameworks, and automated software quality tools. Additionally, Eclipse hosts projects[7] that cover runtimes; static and dynamic languages; thick-client, thin-client, and server-side frameworks; and modeling and business reporting tools. Due to the wealth of the entire Eclipse ecosystem, we selected Eclipse as the

[5] A feature is a group of Eclipse plug-ins.
[6] http://marketplace.eclipse.org/
[7] http://projects.eclipse.org/

implementation technology of this thesis. Below, in Sections 2.3.2, 2.3.3, 2.3.4, and 2.3.5 we describe the specific frameworks and tools that we used.

2.3.2 Eclipse Modeling

The Eclipse Modeling project[8] focuses on the evolution and promotion of model-based development technologies within the Eclipse community by providing a unified set of modeling frameworks, tooling, and standard implementations. Inside this project, we can find, among others, the following tools:

Eclipse Modeling Framework (EMF). EMF is a modeling framework and code generation facility for building tools based on a structured data model. From a metamodel specification, EMF produces a set of Java classes that implement the metamodel, along with a set of adapter classes that enable viewing and command-based editing of models; EMF also produces a tree-based editor. For the specification of metamodels, EMF provides the Ecore language, which claims to be a subset of the MOF standard.

Graphical Modeling Framework (GMF). GMF implements a model-driven approach for the generation of graphical editors in Eclipse. By defining a tooling, graphical, and mapping model, GMF can generate a fully functional graphical editor for creating models based on EMF.

Graphiti. Graphiti shares the same objective as GMF. It is an Eclipse-based graphics framework that enables rapid development of graphical editors. One important difference between Graphiti and GMF is that Graphiti can deal with any Java-based object on the domain side, not only EMF-based domain models. Another difference is that GMF follows a model-driven generative approach, while Graphiti provides a Java API for the manual programming of graphical editors.

Textual Modeling Framework (XText). Xtext is a framework for the development of textual DSLs. In XText, DSLs are specified using a grammar language. From a grammar specification, XText creates a parser, a metamodel (implemented in EMF) as well as a full-featured textual editor.

[8] http://www.eclipse.org/modeling/

Xpand. Xpand is a statically-typed template language for implementing M2T transformations. For the development of Xpand templates, Eclipse provides an editor with features like syntax coloring, error highlighting, navigation, refactoring, and code completion. Xpand was originally developed as part of the openArchitectureWare project[9] before it became a component under Eclipse.

Atlas Transformation Language (ATL). ATL is a hybrid M2M transformation language that allows both declarative and imperative constructs to be used in transformation definitions [117]. ATL is supported by a set of development tools such as an editor, a compiler, a virtual machine, and a debugger.

Summary and Discussion

The CAME environment that has been developed in this thesis (MOSKitt4ME) makes intensive use of the technologies that are described above. Specifically, EMF, GMF, and Graphiti lay the foundations of the tools that allow method engineers to create method models. Furthermore, Xpand is the language that we used to implement the M2T transformation that enables the generation of integrated software environments from the method models.

In addition to all of this functionality, the above modeling frameworks also allow MOSKitt4ME to support the construction of tools for software development (in a similar way to the metatools that are provided by metaCASE environments). For instance, GMF and Graphiti support the construction of graphical editors, while EMF and XText allow users to develop tree-based editors and textual editors, respectively. On the other hand, ATL enables the development of M2M transformations, while Xpand supports the implementation of code generators (i.e., M2T transformations). All of these types of tools (i.e., textual/tree-based/graphical editors, code generators, and M2M transformations) can be integrated in software environments that are adapted to particular methods. Thereby, MOSKitt4ME supports the method implementation phase of the Method Engineering lifecycle.

[9]http://www.openarchitectureware.org/

2.3.3 Eclipse Process Framework

The Eclipse Process Framework (EPF) project[10] provides tools for software process engineering as well as extensible process content for a wide range of processes supporting different approaches for software development (such as iterative, agile, and incremental). The major tool that is distributed as part of the EPF project is called EPF Composer.

The **EPF Composer** is an open-source SPEM 2.0 editor for process engineers and project managers to implement, deploy, and maintain software processes for organizations or individual projects. The most relevant capabilities provided by the EPF Composer are the following:

Method Authoring. The EPF Composer allows users to capture best practices as a set of reusable method building blocks that are defined in SPEM 2.0 (e.g., roles, work products, tasks, and guidance). The properties and relationships of these elements can be defined through forms and rich-text editors.

Process Authoring. Reusable method building blocks can be organized into processes by defining Work Breakdown Structures. The EPF Composer also supports the construction of reusable process chunks through capability patterns. Structural information can be edited with graphical and non-graphical editors.

Library Management. Method libraries enable persistency and flexible configuration management as well as content interchange for distributed client-server implementations. Method and process content can be packaged into plug-ins allowing simple distribution, management, and extensibility of content.

Configuring and Publishing. The EPF Composer allows users to publish process configurations, which are subsets of process elements. During the publishing process, the EPF Composer resolves the relationships between the process elements; then, it generates a set of HTML pages that contain links representing these relationships to make the resulting website easy to navigate.

[10] http://www.eclipse.org/epf/

Summary and Discussion

The EPF Composer has been integrated in MOSKitt4ME to support the construction of SPEM 2.0 models. This endows MOSKitt4ME with the capabilities that are mentioned above: method and process authoring, library management, and method configuration/publishing, among others. Additionally, we have extended the EPF Composer with a repository client that allows users to connect to FTP repositories in order to store, retrieve, and integrate reusable method parts (which are instances of one or more classes of the SPEM 2.0 metamodel).

2.3.4 Activiti

Activiti[11] is a light-weight open-source workflow and Business Process Management platform that is targeted at business people, developers, and system administrators. Its core is a super-fast and rock-solid BPMN 2.0 process engine for Java. Activiti is not an Eclipse project, but it provides an eclipse-based component for the creation of BPMN 2.0 models that can be directly executed in the process engine.

Specifically, in this thesis we have used two of the components that are distributed as part of the Activiti project:

Activiti Engine. This component is the heart of the Activiti platform. It is a Java process engine that runs BPMN 2.0 processes natively. The Activiti Engine has several positive features. Some features that deserve extra highlighting are the following: it runs on any Java environment; it is easy to get up and run; it is rock-solid and extremely fast; and it supports the execution of pieces of custom Java code or scripts upon the occurrence of certain process events.

Activiti Designer. The Activiti Designer is an Eclipse-based component that implements a graphical editor that supports modeling, testing, and deployment of BPMN 2.0 processes. It also has built-in support for the Activiti-specific extensions that enable the use of the full potential of the Activiti engine. Additionally, the Activiti Designer offers the possibility to extend

[11] http://activiti.org/

its default functionality, allowing an easy adaptation to specific business domains or use cases.

Summary and Discussion

One noteworthy capability of Activiti is that it supports all aspects of Business Process Management, including non-technical aspects (such as the analysis and modeling of business processes) as well as technical aspects (such as the creation of software support for business process execution). In MOSKitt4ME, we leverage this capability of Activiti to achieve a twofold objective. The first goal is to provide modeling support for BPMN 2.0; to achieve this goal, we integrated the Activiti Designer into MOSKitt4ME. The second goal is to support the method execution phase of the Method Engineering lifecycle; to this end, we extended the M2T transformation that supports the generation of CASE environments so that it can integrate the Activiti Engine into these tools.

2.3.5 MOSKitt

As Section 1.5 describes, the MOSKitt project[12] started in 2007 as a jointly work between the PROS research center and the CITMA and its main goal was to develop a RCP product that supports the gvMétrica method. The result was an open-source CASE environment called MOSKitt.

One important characteristic of MOSKitt is that, since it is built on Eclipse, it implements the Eclipse plug-in architecture, and, therefore, MOSKitt can be easily reconfigured and/or extended with new tools. Nonetheless, MOSKitt already incorporates much built-in functionality that supports different aspects of the software development lifecycle. The frameworks and tools that are included in MOSKitt can be categorized as follows:

Software Development. MOSKitt incorporates a wide range of tools that enable the analysis, design, and implementation of software systems. Some examples of these tools are graphical editors (e.g., for defining UML 2.0 models, graphical user interfaces, and database schemas) and M2T transformations

[12] http://www.moskitt.org/

for the automatic generation of documentation and source code (in different languages such as Java and PHP). MOSKitt also includes M2M transformations (e.g., for transforming UML 2.0 models into database schemas) and form-based editors for creating different types of models (e.g., glossaries, configuration models, and data dictionaries).

Method Support. To support the gvMétrica method, MOSKitt includes the Dashboard component. This component provides (1) a graphical editor for the specification of software development processes, and (2) an interpreter that allows users to monitor the state of process instances. The specification language that is implemented in the Dashboard editor supports the definition of the following elements: tasks; their input and output resources; actions that are associated to the tasks; and dependencies between tasks. This is inspired by the dashboard provided as part of GMF, which provides assistance during the creation of graphical editors.

Metatools. MOSKitt incorporates metatools that allow users to develop tools for software development. One example of these so-called metatools is the FEFEM framework. This framework enables the rapid development of form-based editors that manipulate EMF models. Specifically, FEFEM implements a series of patterns that usually occur when developing this kind of editors (e.g., a textbox for editing properties of type String); thus, editors are built via pattern composition. In addition to FEFEM, MOSKitt also benefits from similar frameworks that are provided by the Eclipse community; for instance, XText and GMF, which allow users to develop textual and graphical editors, respectively (see Section 2.3.2).

Technological Infrastructure. The MOSKitt technological infrastructure provides four main components. First, a transformation manager, which offers an intuitive Graphical User Interface that facilitates the definition and execution of model transformations. Second, a model synchronization mechanism, which automatically updates the models that are output of model transformations when the input models are manually modified by the user. Third, a tree-based model navigator, which extends the functionality provided by the Eclipse Common Navigator Framework (e.g., the MOSKitt navigator customizes the element labels based on the user's selection). Finally, MOSKitt provides the MDT Common component, which implements functionality that is shared by all of the MOSKitt graphical editors (e.g., drag and drop; filtering of model elements; copy and paste; and storage of multiple diagrams in a single file).

Summary and Discussion

All of the tools that are summarized above turn MOSKitt not only into a CASE environment (that supports a single development method), but also into an extensible platform for building CAME and CASE environments. For this reason, we used MOSKitt as the base platform for implementing the CAME environment that is presented in this thesis: MOSKitt4ME.

2.4 Conclusions

This chapter provides a general overview of various concepts, domains, techniques, frameworks, and tools that are related to the work presented in this thesis. First, the chapter provides a detailed definition of Method Engineering, which is the domain that is the focus of our research work. Method Engineering is a challenging research field. One of its major goals is to facilitate the process of software development, and, therefore, it may contribute to significantly improve the quality of software systems in a near future. In this context, MDE seems to embody a suitable technological background to efficiently face the inherent complexity of Method Engineering. Due to the potential relevance of MDE in the context of Method Engineering, this chapter also summarizes some of the most significant MDE techniques: metamodeling, model transformations, and models at runtime. Finally, the chapter surveys some eclipse-based technologies that are of special significance in this thesis. These technologies, together with the use of MDE, allowed us to design a model-driven Method Engineering approach as well as to implement a supporting CAME environment, both of which are presented in this document. Before we present our work, Chapter 3 provides a state-of-the-art review of the more relevant contributions in the field of Method Engineering.

Chapter 3

State of the Art

Method Engineering has an extensive and disparate history. During the last decades, the Method Engineering community has performed a lot of research work, aiming to overcome the challenges that Method Engineering poses. In this chapter, we review this research work, a task that was last undertaken by Henderson-Sellers *et al.* in [3] and later by Kuhrmann *et al.* in [11].

Specifically, the present chapter focuses on the Method Engineering approaches that are closely related to the thesis work. As Chapter 2 describes, this thesis focuses on three phases of the Method Engineering lifecycle: design, implementation, and execution. For this reason, the present chapter reviews the most relevant approaches that support at least one of these phases. By analyzing these approaches, we aim to illustrate their main limitations, and, thus, disclose the potential contributions of this thesis.

Prior to the state-of-the-art review, this chapter defines a set of distinctive properties that allow us to characterize the selected approaches. These properties are described in Section 3.1. Then, in Section 3.2, we use the properties to analyze the approaches individually. Finally, in Section 3.3, we analyze the approaches in conjunction in order to draw the conclusions of the review.

3.1 Properties to Analyze Method Engineering Approaches

In this section, we define four properties that allow us to characterize the approaches that are reviewed in this chapter. The main goal of these properties is to facilitate the evaluation of the approaches with respect to the problems identified in this thesis: high complexity, incomplete support to the lifecycle of methods, weak process support, and limited software support (see Section 1.2). To this end, each property offers information that relates to one of these problems. For instance, the "lifecycle coverage" property, when applied to a specific approach, offers information about the Method Engineering phases that are supported by the approach. Thus, this property allows us to determine the extent to which the approach experiences the second problem (i.e., incomplete support to the lifecycle of methods). The four properties that we define in this section are the following:

Specification language(s). This property refers to the DSLs that are defined and/or applied in a Method Engineering approach. As Niknafs *et al.* emphasize in their evaluation framework for Process Modeling Languages [21], this property involves a trade-off between automation and complexity. On the one hand, the more formal rigor is applied in a language, the more automated support is possible (e.g., for method execution). On the other hand, the more formalism is added, languages get less understandable (i.e., they become more complex). In order to strike a perfect balance between automation and complexity, formal specification languages can be used as long as their complexity is reduced through usable and easy-to-use graphical notations.

We divide the specification language(s) property into four subproperties:

- *Name*: this property indicates the name(s) of the language(s).
- *Type*: this property indicates whether the language(s) are textual, graphical, or both. The type of a language is generally determined by the underpinning formalism of the language (e.g., attribute grammars are used to define textual languages, while metamodels are typically used to define graphical languages). With respect to complexity,

textual languages are commonly considered to be more complex than graphical languages, mainly due to their lower level of abstraction.

- *Formality level*: it gives an approximate indication of the complexity of the language(s). Textual languages with a high level of formality (e.g., languages with a high number of formal rules and constraints) are considered to be highly complex. In contrast, the languages that offer high-level constructs and simple graphical notations are considered to be easy to use and to have a low level of formality.
- *Size*: this property also gives an indication of the complexity of the language(s). If a language is big (i.e., it has a high expressive power), then it is more likely that the language yields large specifications, which are usually complex and hard to understand. In contrast, small languages typically produce simpler specifications, but this comes at the expense of losing expresiveness.

Lifecycle coverage. This property indicates the phases of the Method Engineering lifecycle that are covered by the Method Engineering approach. As Figure 2.3 shows, the lifecycle of a software development method comprises the following generic phases: analysis (MA), design (MD), implementation (MI), execution (ME), and monitoring (MM). In summary, the first phase is to analyze the requirements of the method. When these requirements are fully analyzed, the method is designed and implemented. Subsequent executions of the method can be monitored to obtain feedback that is fed to the first phase in order to start another Method Engineering cycle. As it is stated in chapter 1, our research focuses on three phases (i.e., design, implementation, and execution); therefore, we do not consider the phases of analysis and monitoring in the literature review that is presented in this chapter.

Perspective. This property concerns the product and process parts of methods. If the Method Engineering approach emphasizes the specification of the structure of products and deliverables as well as the specification of how these products must be created and manipulated, then the approach is product-oriented. In contrast, if the approach focuses on the specification and enactment of processes, then the approach is process-oriented. Ideally, a Method Engineering approach should be able to represent methods from both the product perspective and the process perspective.

Software support. This property indicates whether the Method Engineering approach is supported by a software tool. In the Method Engineering literature, we can find two types of tools: CAME environments and metaCASE environments. While CAME environments, in their ideal form, provide software support for all the conceptual aspects of methods (such as the analysis of the method requirements or the conceptual specification of the method product and process parts), metaCASE environments focus on the technical aspects; that is, metaCASE environments are oriented towards the construction of CASE environments that provide complete software support for the methods (see Section 2.1.3). Ideally, a software tool that supports a Method Engineering approach will share features from both CAME and metaCASE technology; that is, the software tool will allow method engineers to fully specify software development methods as well as to build the supporting CASE environments.

We divide the software support property into the following four subproperties:

- *Name*: this property indicates the name of the tool.
- *Type*: this property indicates whether the tool is a CAME environment, a metaCASE environment, or it shares features from both types of technologies.
- *Use*: this property indicates whether the tool is a research prototype or rather it has traversed academic boundaries (i.e., it has been used for commercial purposes).
- *Automation level*: this property gives an approximate indication of the degree to which the software tool automates Method Engineering activities.

3.2 Method Engineering Approaches

In this section, we analyze the Method Engineering approaches of our review using the four properties that are defined in Section 3.1. Specifically, for each approach,

TABLE 3.1: Template for approach classification

Approach name		
Specification language(s)	**Name**	Name(s) of the language(s)
	Type	{Textual, Graphical}
	Formality level	[Low | Medium | High]
	Size	[Small | Medium | Big]
Lifecycle coverage		{MD, MI, ME}
Perspective		{Product, Process}
Software support	**Name**	Name of the tool
	Type	{CAME, MetaCASE}
	Use	{Research, Commercial}
	Automation level	[Low | Medium | High]

we present an in-depth description first; then, in a section that is called "Analysis of the Proposal", we present the main limitations of the approach together with a summary that follows the template that is shown in Table 3.1. The left side of the template contains the properties that we propose for analyzing Method Engineering approaches (see Section 3.1); the right side shows the possible values of these properties. These values can be of three different types: string, enumeration, and collection. String properties (i.e., name) can take as value any string of characters. On the other hand, enumerations (i.e., formality level, size, and automation level) can take as value any of the elements that are contained in the set denoted by the [ ] symbols. Unlike strings and enumerations, collections (i.e., type, lifecycle coverage, perspective, and use) can take multiple values; specifically, the possible values of the properties of type collection are enclosed within { } symbols. When one of the properties that are shown in Table 3.1 does not apply to the approach under study (either because it is not supported or the required information is not published), we set the value "x" to the property.

The Method Engineering approaches that are analyzed in this section are divided into three categories: assembly-based, paradigm-based, and configuration-based. We determined these categories by examining the Method Engineering literature [3, 6, 9, 118, 119], which generally classifies approaches in one of the following five categories: assembly-based, paradigm-based, configuration-based, ad-hoc, and extension-based. Note that we consider a reduced set of three categories. This is because ad-hoc Method Engineering involves the construction of methods from scratch [3, 120, 121], and, for this reason, this category has been somewhat neglected in the literature; furthermore, extension-based approaches can be considered to be a specific type of configuration-based Method Engineering [4, 33]. For these reasons, the Method Engineering approaches of our review are divided into three categories, which are presented in Sections 3.2.1, 3.2.2, and 3.2.3, respectively. Within these sections, the approaches are ordered chronologically according to their year of introduction. Prior to the three sections, we briefly describe below the process that we followed to select the papers that are considered in our literature review.

The selection process

To perform the review that is presented in this chapter, we followed the guidelines for performing literature reviews proposed by Webster *et al.* in [122]. As these guidelines suggest, we performed three steps to determine the source material of our review. First, we searched leading journals and also conference proceedings with a high reputation for quality. Second, we went backwards in time by reviewing the citations of the identified articles; thus, we found additional articles, which in general predated the articles that had been identified in the first step. Finally, we went forward by using search engines (e.g., google scholar[1]) and scientific databases (e.g., science direct[2]); these tools allowed us to identify articles that cite the articles that had been identified in the previous two steps.

After these three steps were carried out, we applied various exclusion criteria. Specifically, we excluded the papers that: (1) were not presented entirely in the English language; (2) did not represent scientific papers but rather some type of non-peer-reviewed publication, such as technical reports, book chapters, or proceedings' prefaces; and (3) presented some kind of review (e.g., surveys or mapping studies) rather than outcomes of technical research work.

3.2.1 Assembly-based

Assembly-based Method Engineering fosters the construction of reusable method parts, which are later assembled to obtain methods that are adapted to the context

[1] http://scholar.google.es/
[2] http://www.sciencedirect.com/

of use. In the literature, different terms have been proposed to name these so-called method parts; the most common terms are method fragments, method chunks, method components, and method blocks. All of these types of method parts are generally defined as “small reusable portions of methods, either methods that already exist or methods-to-be” [26].

Typically, method parts are stored in a repository that is called method base. One important requirement for the method parts that are stored in a method base is that their format must be standardized; otherwise, the interoperability between method parts will be restricted. One common way of facilitating interoperability is to ensure that each method part conforms to the same higher-level definition; for instance, the one given by a metamodel.

Some of the benefits that are provided by assembly-based Method Engineering approaches are: increased reuse, modular methods, and the possibility to build repositories of method knowledge. In the following subsections, we present some of the most relevant assembly-based approaches.

3.2.1.1 Brinkkemper *et al.* Approach

The approach of Brinkkemper *et al.* [8, 123] relies on the notion of method fragment, which was coined in 1994 [124] and later popularized in 1996 [5]. A **method fragment** can be defined as “a description of a software development method, or any coherent part thereof”. Method fragments are classified according to three different dimensions: perspective, abstraction level, and granularity layer. The perspective dimension comprises product and process. Product fragments represent, e.g., deliverables, models, or diagrams; process fragments represent stages, activities, and tasks to be carried out. On the other hand, the abstraction dimension constitutes the conceptual level and the technical level. Conceptual fragments are descriptions of methods or parts thereof; technical fragments are implementable specifications of the operational part of methods (i.e., the software tools). Finally, the granularity layer represents a level of decomposition within the method. For instance, from a process perspective, methods can be decomposed into stages, which are further partitioned into activities and individual steps. From the product perspective, the final system would be at the top of the tree; this system can be further decomposed into, e.g., deliverables, model components,

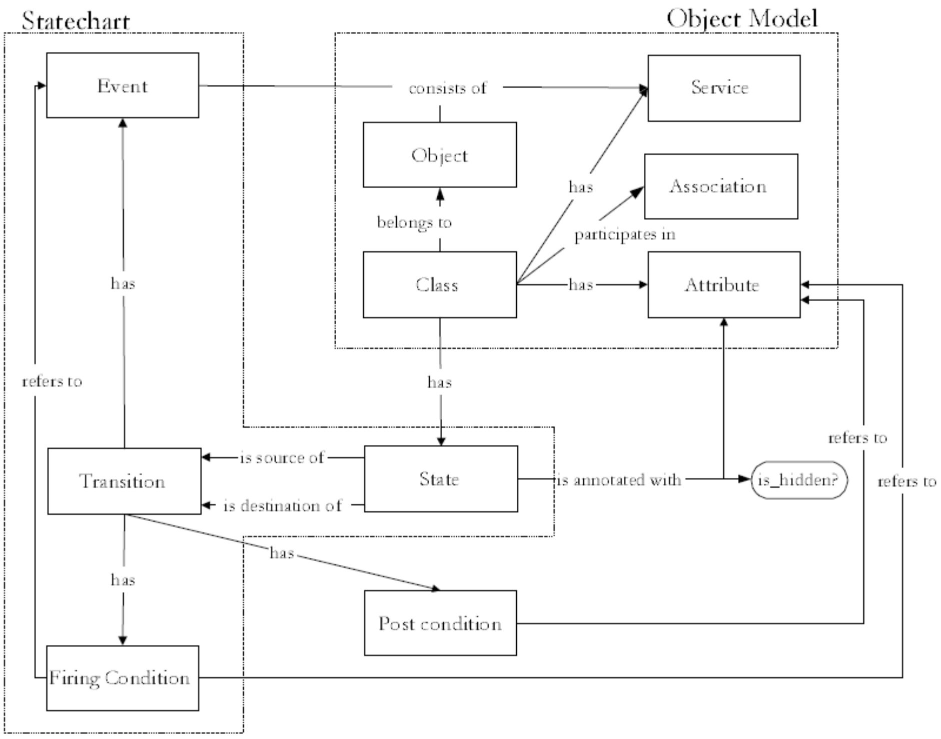
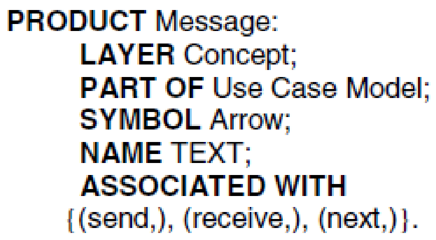


FIGURE 3.1: Assembly of product fragments [123]

and concepts. As an example, Figure 3.1 shows the concepts of two (conceptual) product fragments: the Object Model and the State Chart. These fragments have been assembled to obtain the Object Chart product fragment.

In order to carry out the assembly of method fragments, Brinkkemper *et al.* [8, 123] propose a set of **assembly rules** that are formalized by means of first order logical formulas. These rules establish how to connect method fragments through newly introduced associations and/or concepts. An example of assembly rule is: "if we add new associations, the two method fragments to be assembled should participate in them". These rules impose constraints in the assembly process, allowing method engineers to obtain meaningful methods. Most of the constraints that Brinkkemper *et al.* define are syntactical, but they emphasize the need for semantical constraints as well. Since semantical constraints require the formalization of the fragment semantics, Brinkkemper *et al.* propose the use of an anchoring system. Using an anchoring system, method fragments can be



```
PRODUCT Message:
    LAYER Concept;
    PART OF Use Case Model;
    SYMBOL Arrow;
    NAME TEXT;
    ASSOCIATED WITH
   {(send,), (receive,), (next,)}.
```

FIGURE 3.2: Example of product fragment in MEL [126]

anchored to unambiguous concepts that define them. In Brinkkemper's proposal, these concepts are defined by means of an ontology that is called Methodology Data Model (MDM) [20, 61]. Some examples of the concepts that are proposed by MDM are: Activity, Goal, Group, Problem, Requirement, Role, and System.

The proposal of Brinkkemper *et al.* is supported by Decamerone [20, 125], a CAME environment that allows method engineers to define methods using a language that is called "Method Engineering Language" (MEL) [126]. MEL is a textual language that supports both the product and process aspects of methods at varying granularity levels and is founded on first order predicate logic. It also provides operations to insert and remove fragments in and out of the method base. To support MEL, Decamerone provides dedicated editors and a MEL interpreter. Additionally, Decamerone allows method engineers to obtain project-specific CASE environments since it is built on top of a metaCASE system called Maestro II. The project-specific environments that are obtained by Decamerone contain a product repository, a process manager, and a set of editors that enable system specification. A simplified example of the description of a product fragment using MEL is given in Figure 3.2. For a more in-depth summary of Decamerone see [41].

Analysis of the Proposal

To our understanding, the approach of Brinkkemper *et al.* is one of the most complete, especially in terms of lifecycle coverage and method perspectives. Furthermore, it provided a high number of innovations and contributions at the time it was presented. However, it still presents some limitations, being the following the most relevant:

- *High complexity*: MEL is a textual language that offers a high number of primitives, which makes the language difficult to learn. Additionally, Maestro II is an evolution of an environment that has its roots in the seventies. This is reflected in the character-based user interface, which complies with no standard whatsoever. The screens of Decamerone look clumsy and hard to use when compared to current sophisticated graphical interfaces [20].

- *Limited technical support*: MEL is weak regarding the representation of the technical side of methods [20]. For instance, the CASE environments that are obtained using Decamerone can only contain editors; other tools such as model transformations cannot be included.

- *Limited process executability*: MEL is less suitable in situations where process execution should be possible . Only using the detour of process engine generation, it is possible to execute a set of process managers [20]. Process managers are technical fragments that guide the CASE user through the development process. These managers connect products with tasks, and, also, they are able to start the execution of the CASE editors as well as other process managers.

- *Limited workflows*: MEL only supports the specification of two types of relationships (i.e., precedence and conditional precedence) between process fragments; more complex sequencing behavior (such as the behavior that is supported by the gateways of BPMN 2.0) is neglected. Thus, the resulting processes may be too limited for software engineers working on real development projects.

- *Lack of producer fragments*: even though the method fragment notion addresses both the product and process aspects of methods, the human aspects can only be specified by means of the “Producer” attribute of product fragments. The lack of producer fragments restricts the reusability that can be achieved with respect to these kind of method elements.

Table 3.2 summarizes the relevant information of Brinkkemper’s approach according to the template that is presented in Table 3.1.

Table 3.2: Classification of Brinkkemper *et al.* approach

Brinkkemper *et al.* approach		
Specification language(s)	**Name**	MEL
	Type	Textual
	Formality level	High
	Size	Big
Lifecycle coverage		MD, MI, and ME
Perspective		Product and Process
Software support	**Name**	Decamerone
	Type	CAME and MetaCASE
	Use	Research
	Automation level	Medium

3.2.1.2 Prakash Approach

The approach of Prakash [7] is based on a three-level architecture where methods are defined as collections of method blocks. The three levels of this architecture are: the generic level, the method-independent level, and the method level. In the generic level, the focus is the intrinsic notion of the method (i.e., what it can do, what it is, etc.). This level is the representation of the method in its most abstract form; thus, the generic level is metamodel-independent. On the other hand, the method-independent level involves the instantiation of the generic view of the method as a metamodel that provides a metamodel-specific view. This metamodel provides concepts that are specific instantiations of the corresponding concepts of the generic view. Finally, at the third level, the metamodel is instantiated to yield specific methods.

Figure 3.3 graphically depicts the generic view of methods. It is at this point important to clarify that, in Prakash's proposal, "methods" are not "software development methods", but rather specification languages (e.g., the Entity-Relationship model [127]) that can be used to represent particular features of software systems (e.g., database schemas). Having clarified this issue, one can observe in Figure 3.3 that methods are composed of method blocks. A **method block** represents a decision that the software engineer can make (e.g., creating an entity or deleting an attribute) and it is defined as a pair <*objective*, *approach*>. The objective of a method block tells us what the block tries to achieve. For objectives to be well-defined, it is necessary to know what can be done (the *process type*) on what

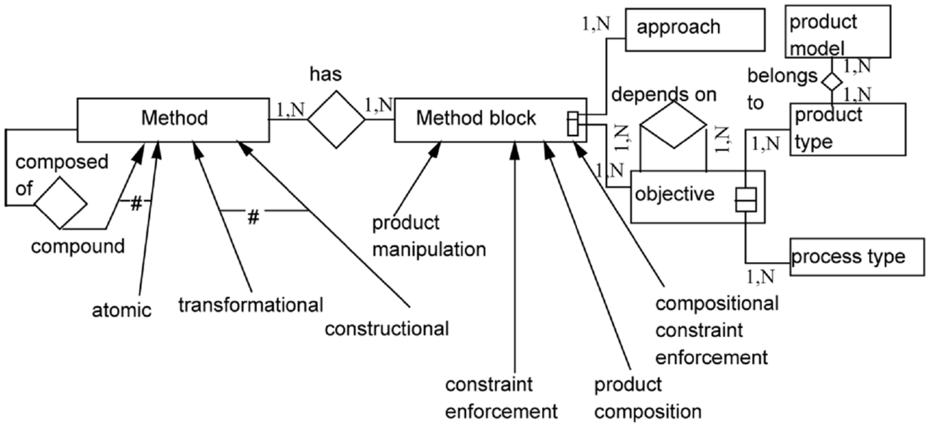


FIGURE 3.3: The generic view of methods [7]

(the *product type*). Two examples of objectives in the Entity-Relationship (ER) model are "create entity" and "delete attribute". In these examples, "create" and "delete" are the process types and "entity" and "attribute" are the product types, which belong to one or more *product models* (in this case, specific ER models). Since the objectives need to be realized (or, in other words, the method blocks/decisions have to be executed), the *approach* identifies the manner in which this is done (e.g., by means of a button of the toolbar of the CASE environment). When an objective is executed, the execution of other objectives may be enabled/forbidden. These relationships between objectives are called *dependencies*.

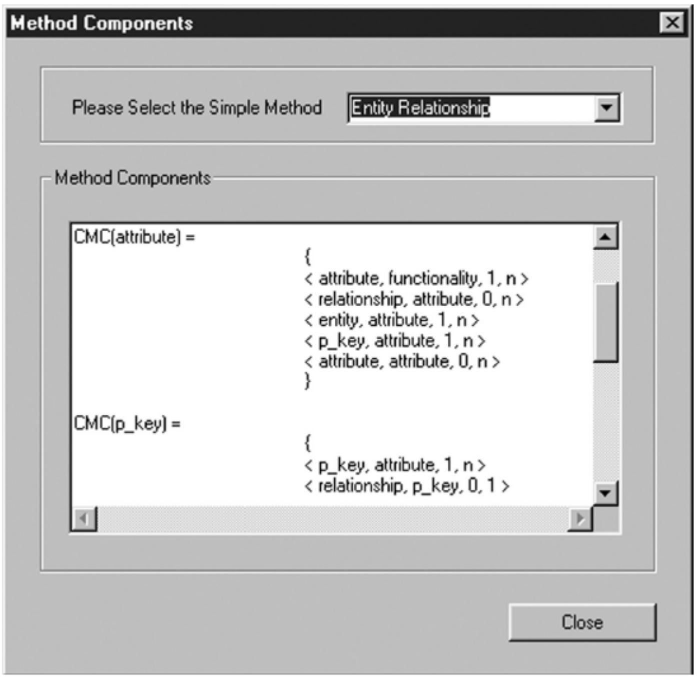
There are two types of methods that can be built using method blocks: transformational and constructional. A *transformational* method is used for transforming a product, expressed in one or more product models, into a product of other product model(s); for instance, a transformational method can be used to transform the entities/relationships of a ER model into classes/associations of a UML model. On the other hand, a *constructional* method is used whenever a new product is to be constructed.

Moreover, any method, whether transformational or constructional, can be atomic or compound. An *atomic* method deals only with those products that are expressed in exactly one product model. In contrast, a *compound* method is composed from other simpler methods.

Finally, there are four kinds of method blocks: *product manipulation, constraint enforcement, product composition, and compositional-constraint enforcement.* The first type deals with the creation, edition, and deletion of product types, while the second type deals with the enforcement of constraints when these actions are performed. The last two types deal, respectively, with the establishment of correspondences between the products of a compound method and the enforcement of composition constraints.

The approach presented up to this point is extended in [128] with the concept of **method dynamics**, as opposed to the **method statics** shown in Figure 3.3. Specifically, the dynamic aspects of a method identify the manner in which the method can be used to develop products. In other words, method dynamics deal with the selection of the method blocks to be executed. With respect to the method dynamics, Prakash introduces new concepts such as *development program*, which is the sequence of decisions (i.e., the sequence of executed method blocks) followed by software engineers for product creation. Since this decision-making process is not defined as part of the method, Prakash's approach can be considered to be fully product-oriented.

With respect to the tool support, Prakash's approach is implemented as a metaCASE environment that is called MERU (which stands for "Method Engineering Using Rules") [38]. In this metaCASE environment, methods are defined in three sequential steps. First, the method engineer expresses the method requirements using a textual language that is called Method Requirements Specification Language (MRSL). The requirements specification (MRS) represents the generic view of the method. A method analyzer checks the MRS and looks for incompleteness, inconsistency, and non-conformity with a metamodel that is called Method View Model (MVM). Based on this analysis, data is generated to provide guidance to the method engineer on how to improve the MRS. Once the MRS is complete, the second step starts. In this step, the MRS is translated into an instantiation of the MVM metamodel; this instantiation represents the method. Finally, the last step involves the use of the resulting method for the generation of the supporting software environment. As an illustration of MERU, Figure 3.4 shows a screenshot of the tool. This screenshot illustrates how the MRS is defined as an assembly of components developed using MRSL. For a more in-depth outline of MERU, see [41].



Entity Relationship

```
CMC(attribute) =
                        {
                        < attribute, functionality, 1, n >
                        < relationship, attribute, 0, n >
                        < entity, attribute, 1, n >
                        < p_key, attribute, 1, n >
                        < attribute, attribute, 0, n >
                        }

CMC(p_key) =
                        {
                        < p_key, attribute, 1, n >
                        < relationship, p_key, 0, 1 >
```

FIGURE 3.4: Screenshot of MERU [38]

Analysis of the Proposal

The work of Prakash represents a great effort to formalize and regularize the conceptual framework and underpinning theory of development methods and metaCASE technology. Nonetheless, this work still presents some limitations. The most relevant are the following:

- *High complexity*: the approach of Prakash is complex to apply in real contexts mainly due to the textual nature and high formality of the language supported by MERU.
- *Incomplete lifecycle coverage*: in a similar way to all metaCASE environments, MERU focuses exclusively on the technical aspects of software development methods. That is to say, MERU supports the method implementation phase of the Method Engineering lifecycle (allowing method engineers to define the specification languages to be included in the final software environments), neglecting the method design phase.

TABLE 3.3: Classification of Prakash approach

Prakash approach		
Specification language(s)	**Name**	MRSL
	Type	Textual
	Formality level	High
	Size	Small
Lifecycle coverage		MI
Perspective		Product
Software support	**Name**	MERU
	Type	MetaCASE
	Use	Research
	Automation level	High

- *Fully product-oriented*: Prakash's approach is based on what is called "passive viewpoint". Passive methods identify the set of decisions that can be taken on a given product but leave it to the software engineer to select the one to be executed. The sequence of selections constitutes the process adopted and this process is external to the method; that is, the method does not prescribe any way of working.

Table 3.3 summarizes the relevant information of the approach that is proposed by Prakash according to the template that is presented in Table 3.1.

3.2.1.3 Ralyté *et al.* Approach

The assembly-based approach that is proposed by Ralyté *et al.* [23] relies on the notion of method chunk [129, 130]. A **method chunk** can be defined as "an autonomous and coherent part of a method, which supports the realization of some specific software development activity" [26]. Each method chunk contains a process part (also called guideline) and a product part (which defines the class of products that are obtained as outputs of the chunk). This tight coupling between product and process has been subject to some criticism; for instance, Henderson-Sellers *et al.* [131] argue that there is a potential disadvantage in this process-product linkage since it is neither one-to-one nor unique in real-life scenarios.

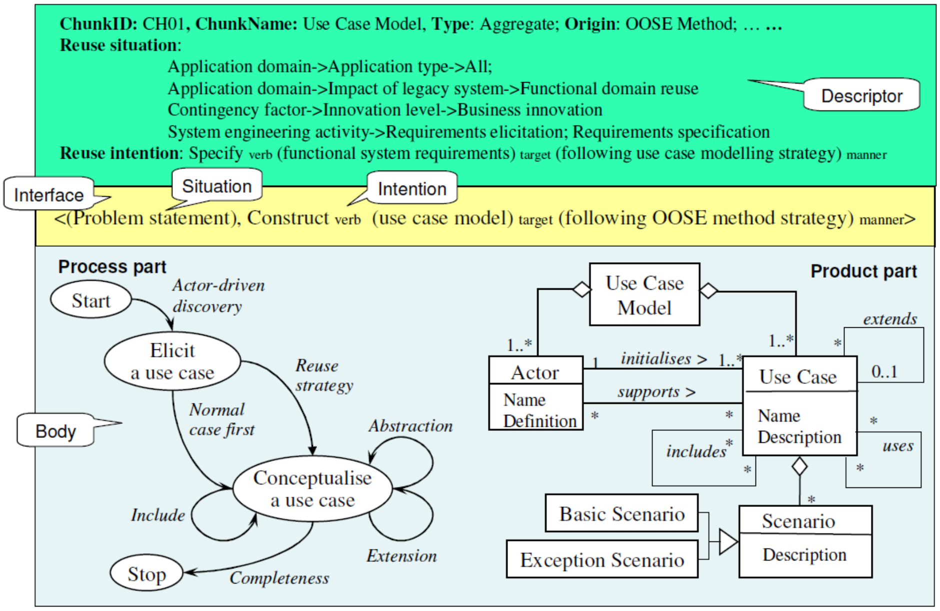

ChunkID: CH01, **ChunkName:** Use Case Model, **Type**: Aggregate; **Origin**: OOSE Method; … …
Reuse situation:
Application domain->Application type->All;
Application domain->Impact of legacy system->Functional domain reuse
Contingency factor->Innovation level->Business innovation
System engineering activity->Requirements elicitation; Requirements specification
Reuse intention: Specify verb (functional system requirements) target (following use case modelling strategy) manner
Descriptor
Interface
Situation
Intention
<(Problem statement), Construct verb (use case model) target (following OOSE method strategy) manner>
Process part
Start
Actor-driven discovery
Elicit a use case
Reuse strategy
Normal case first
Abstraction
Body
Conceptualise a use case
Include
Extension
Stop
Completeness
Product part
Use Case Model
1..*
1..*
Actor
Name
Definition
initialises >
supports >
Use Case
Name
Description
extends
0..1
uses
includes
Basic Scenario
Exception Scenario
Scenario
Description

FIGURE 3.5: Example of method chunk [131]

Figure 3.5 shows an example of method chunk. As the figure shows, method chunks have a *descriptor* that defines the context in which the chunk can be reused. Additionally, chunks have an *interface* that is a tuple $< situation, intention >$ where *situation* is the input of the chunk and *intention* is the goal that the chunk helps to achieve. Finally, the *body* of the chunk defines the product and process parts. While the product part is defined by means of UML, the process part is defined using the Map formalism, which was proposed by Rolland *et al.* in [132].

According to Rolland *et al.* [132], a map is a labelled directed graph with nodes representing *intentions* and edges representing *strategies*. An intention captures the notion of a task to be accomplished whereas a strategy suggests the way in which this goal can be achieved. The core concept of the Map formalism is the *section*, which is a triplet $< I_i, I_j, S_{ij} >$ where I_i is a source intention, I_j is a target intention, and S_{ij} is a strategy that can be followed to achieve I_j from I_i. Thus, a map can be defined as a composition of sections, plus a *Start* and a *Stop* intention [132].

In order to facilitate the assembly of method chunks, Ralyté *et al.* propose in

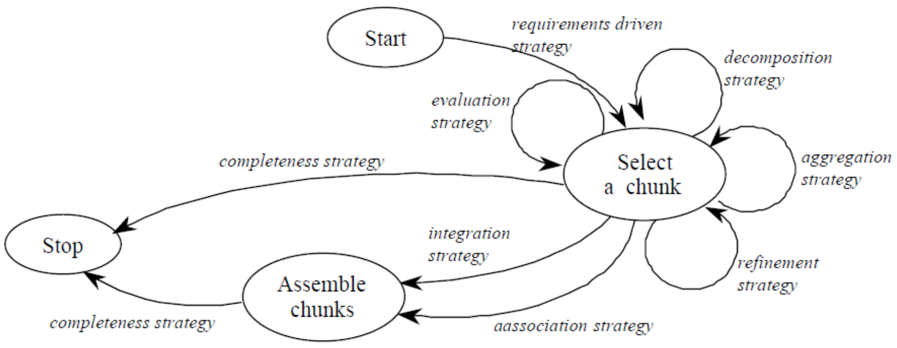


FIGURE 3.6: Process model for chunk assembly [23]

[23] a **generic process model** for guiding method engineers during the assembly process. Figure 3.6 graphically depicts this process model in the form of a map. As the figure shows, Ralyté *et al.* suggest to start the assembly process by selecting the method chunks to assemble. This is indicated by the *select a chunk* intention. To achieve this intention, the only possibility is to follow the *requirements-driven strategy* [133]. This strategy requires that the method engineer specifies the requirements for the method; then, the method chunks matching these requirements are retrieved from the method base. Any time a chunk is retrieved, the process model suggests to validate this candidate chunk by applying the *evaluation strategy*. The evaluation strategy helps in evaluating the degree of matching between the candidate chunk and the requirements; this evaluation is based on similarity measures [23]. If it is necessary to refine the chunk, the decomposition, aggregation, and refinement strategies can be applied. The *decomposition strategy* is relevant when the selected method chunk is an aggregate one having some component parts that may not be required. The *aggregation strategy* is relevant when the candidate chunk partly covers the requirements; this strategy suggests to search for an aggregate chunk containing the candidate chunk based on the assumption that the aggregate chunk might provide a solution for the missing requirements. The *refinement strategy* proposes to search for another chunk satisfying the same intention but providing a set of guidelines richer than those of the candidate chunk.

When at least two chunks have been selected, the method engineer can assemble these chunks following one of two strategies: association or integration. The

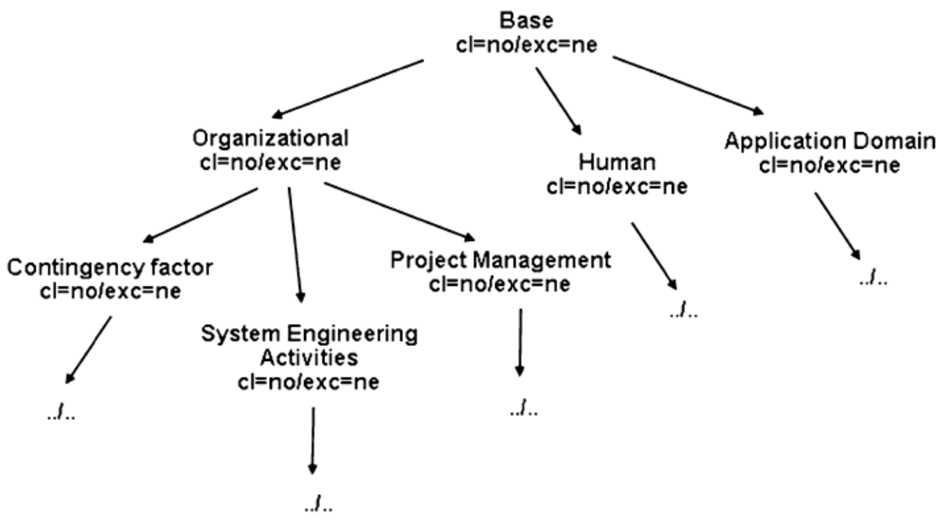


FIGURE 3.7: Reuse frame [134]

association strategy is relevant when the chunks do not have elements in common. This might occur, e.g., when the output product of one chunk is the input product of the other chunk. In this case, the assembly consists in establishing the execution order of the chunks. The *integration strategy* is relevant when the chunks have a similar objective but provide different ways of satisfying it. In this case, the chunks overlap and the assembly consists in merging the overlapping elements. To check whether two chunks overlap, similarity measures can be used [23]. To perform the assembly, Ralyté *et al.* provide in [121] a set of assembly operators. Some examples of these operators are “remove intention”, “add section”, and “connect via association”.

Finally, to check whether the chunk assembly matches the requirements, the method engineer shall use the *completeness strategy.* If the response is positive, then the assembly process ends. In any other case, other chunks have to be selected and assembled to gain the required method completeness.

The proposal of Ralyté *et al.* is extended by Mirbel *et al.* in [134] with the notion of **reuse frame**. The reuse frame is a hierarchical structure that contains knowledge about the reuse context of method chunks and provides criteria for project and software engineer situation characterization; thus, the descriptor of method chunks takes values from the reuse frame, allowing enhanced storage and retrieval

Table 3.4: Classification of Ralyté *et al.* approach (assembly-based)

Ralyté *et al.* approach (assembly-based)			
Specification language(s)	**Name**	UML	Map
	Type	Graphical	Graphical
	Formality level	Medium	High
	Size	Big	Small
Lifecycle coverage		MD	
Perspective		Product and Process	
Software support	**Name**	x	
	Type	x	
	Use	x	
	Automation level	x	

from the repository. As figure 3.7 shows, the reuse frame stores knowledge in terms of *aspects* belonging to aspect *families*, which are successive refinements of three main factors: human, organizational, and application domain. Each aspect of the hierarchy is characterized by two main properties: classified and exclusion. The *classified* property indicates whether direct aspects or subfamilies are classified (cl=yes) or not (cl=no); thus, certain ordering/classification is established between the aspects. On the other hand, the *exclusion* field indicates whether direct aspects or subfamilies are exclusive (exc=e) or not (exc=ne); exclusive aspects cannot be selected for the same project-specific method.

Analysis of the Proposal

Table 3.4 summarizes the relevant information of the approach that is proposed by Ralyté *et al.* according to the template that is presented in Table 3.1. The main limitations of this approach are the following:

- *Lack of software support*: the proposal of Ralyté *et al.* has neither been implemented as a CAME environment nor as a metaCASE tool.
- *Incomplete lifecycle coverage*: in a similar way to most Method Engineering approaches, the approach of Ralyté *et al.* does not address the method implementation and the method execution phases of the Method Engineering lifecycle.

- *Limited workflows*: another limitation concerns the process support. In this approach, the process part of methods is defined when method chunks are assembled, generally by means of precedence relationships that establish their execution order. Thus, the resulting process is quite limited in terms of control flow, since complex behavior (such as the expressed by branching conditions, events, and synchronizations) cannot be defined.

- *Lack of producer fragments*: the method chunk notion does not incorporate the producer aspects of methods, thereby limiting the reusability that can be achieved with respect to this type of method element. Nonetheless, producers are present in the reuse frame, under the "Human" aspect family.

3.2.1.4 OPEN Process Framework

In words of the OPEN Process Framework (OPF) website[3], the OPF [135, 136, 137] is "a practical, public-domain, industry-standard, general-purpose management and engineering process framework that is primarily intended for the object-oriented, component-based development of software-intensive systems". To support the construction of software development methods, the OPF builds on the existing body of knowledge in assembly-based Method Engineering. Specifically, the OPF provides three major components: a metamodel, a repository of reusable method components, and construction guidelines (see Figure 3.8).

The OPF **metamodel**, which has been recently updated to be in conformance with the ISO/IEC 24744 standard [67, 138], provides a clear way for formally representing the OPF method components. It is imperative that each method component in OPF conforms to the OPF metamodel. This implies that newly created method components that extend the repository must also conform with the metamodel.

The OPF **repository** contains a large number of OPF method components having different levels of granularity. According to the ISO/IEC 24744 standard, there are five major types of components in OPF: work units (i.e., pieces of work that must be performed by persons or tools to develop work products), work products (i.e., artifacts that are produced during software development, such as diagrams or

[3] http://www.opfro.org/

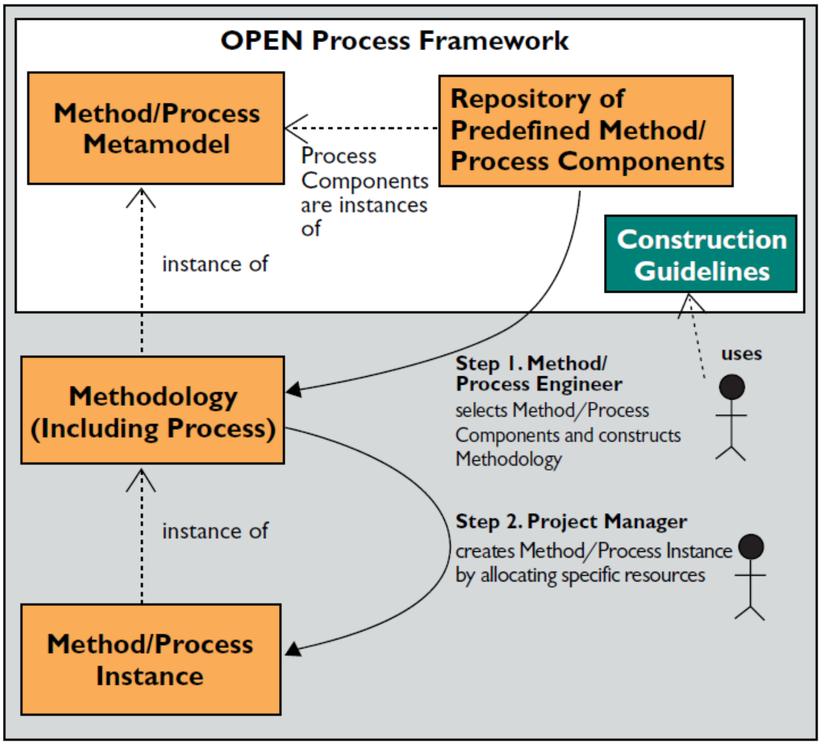
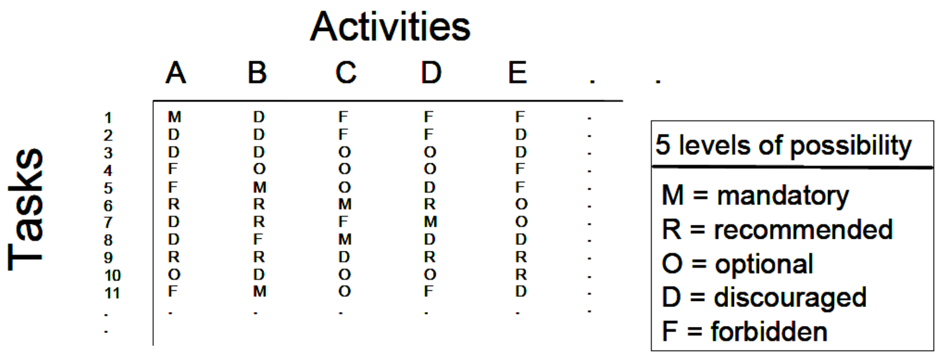

OPEN Process Framework
Method/Process Metamodel
Repository of Predefined Method/ Process Components
Process Components are instances of
instance of
Construction Guidelines
uses
Methodology (Including Process)
Step 1. Method/ Process Engineer selects Method/Process Components and constructs Methodology
instance of
Step 2. Project Manager creates Method/Process Instance by allocating specific resources
Method/Process Instance

FIGURE 3.8: Method construction in OPF [31]

textual documents), producers (i.e., persons or tools that develop work products), languages (i.e., notations used to represent the produced artifacts), and stages (i.e., periods of time used for establishing the overall macro-scale and time-box of sets of cohesive work units).

The **construction guidelines** help method engineers to instantiate the OPF metamodel to create method components as well as to select the best method components from the repository in order to create project-specific methods. As an aid for selecting the best method components, the OPF suggests the use of the deontic matrix approach [135]. A deontic matrix is a two-dimensional matrix where each value represents the likelihood of the relationship between a pair of OPF method components selected from the repository. In this approach, one of five values of possibility is allocated to each matrix element. The possibilities range from mandatory, recommended through optional to discouraged and forbidden [26]. Figure 3.9 shows an example of deontic matrix. The example shows



Activities

Tasks	A	B	C	D	E	.	.
1	M	D	F	F	F	.	
2	D	D	F	F	D	.	
3	D	D	O	O	D	.	
4	F	O	O	O	F	.	
5	F	M	O	D	F	.	
6	R	R	M	R	O	.	
7	D	R	F	M	O	.	
8	D	F	M	D	D	.	
9	R	R	D	R	R	.	
10	O	D	O	O	R	.	
11	F	M	O	F	D	.	
.	.	.	.	.	.	.	
.							

5 levels of possibility
M = mandatory
R = recommended
O = optional
D = discouraged
F = forbidden

FIGURE 3.9: An example of deontic matrix [3]

a two-dimensional relationship between tasks and activities. For each combination of activities and tasks, we can assess the likelihood of the occurrence of the combination using five levels of possibility (which are labelled as M, R, O, D, and F, respectively).

Analysis of the Proposal

In summary, we believe that OPF represents a great contribution that helps pave the way from academic research to the industrial practice of Method Engineering. The limitations that we find are the following:

- *Lack of software support*: this approach has neither been implemented as a CAME environment nor as a metaCASE tool. The OPF should evolve from a simple repository to a fully functional software product.

- *Incomplete lifecycle coverage*: this approach does not address the method implementation and the method execution phases of the Method Engineering lifecycle.

- *Limited workflows*: while the ISO/IEC 24744 standard does support the specification of the process part of methods, it provides limited support for the definition of processes with a complex workflow. In ISO/IEC 24744, process elements are defined as work units, which are allocated in stages. The workflow of the process is established by the stages, which can only be associated by means of precedence relationships. Aharoni *et al.* also highlight this problem in [139] and suggest enriching processes by means of

Table 3.5: Classification of the OPEN Process Framework

OPEN Process Framework		
Specification language(s)	**Name**	ISO/IEC 24744
	Type	Graphical
	Formality level	Medium
	Size	Big
Lifecycle coverage		MD
Perspective		Product and Process
Software support	**Name**	x
	Type	x
	Use	x
	Automation level	x

an extension of the stage concept. This extension allows work units to be combined within stages in more meaningful ways than simple inclusion (e.g., concurrently or iteratively).

- *Limited process executability*: another limitation of the ISO/IEC 24744 standard refers to the lack of formalization of the process execution semantics. This limitation hinders the execution (by means of a process engine) of the processes that are defined by assembling OPF components (since these components are compliant with the ISO/IEC 24744 standard).

Table 3.5 summarizes the relevant information of OPF according to the template that is presented in Table 3.1.

3.2.1.5 Method Editor

In 2003, Saeki [140] presented a CAME environment that is called Method Editor. This CAME environment builds on some of the concepts and techniques that were previously introduced by Decamerone. For instance, Method Editor applies assembly-based Method Engineering techniques and is based on the concept of method fragment.

Figure 3.10 shows the general overview of Method Editor. As the figure shows, the major component of this CAME environment is a method editor that allows

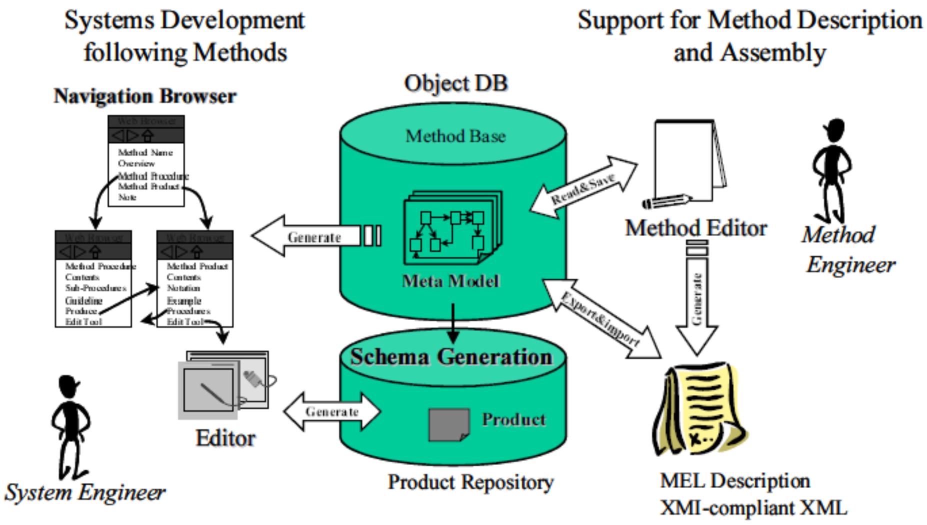


FIGURE 3.10: Overview of Method Editor [140]

method engineers to manipulate the method fragments that are stored in the method base as well as to assemble these fragments to obtain new methods. This editor is a kind of diagram editor that allows method engineers to easily edit method fragments and also to generate, import, and export textual specifications in MEL [126]. For the creation and manipulation of fragments, the editor takes advantage of UML as its metamodeling technique. Class diagrams are used for the specification of product fragments, while process fragments are described by means of activity diagrams. The description of a newly built method is used for generating a software environment supporting the method. This environment incorporates diagram editors, a product repository, and a navigation browser. The diagram editors allow software engineers to create the method products, which are stored in the product repository. The navigation browser provides browsing pages that guide software engineers through the process of software development. While the pages are being browsed, the corresponding diagram editors are automatically invoked.

The most recent version of Method Editor is extended by means of a version control system [141], thereby supporting version control and change management of methods and method fragments.

Table 3.6: Classification of Method Editor

Method Editor		
Specification language(s)	**Name**	UML
	Type	Graphical
	Formality level	Medium
	Size	Big
Lifecycle coverage		MD, MI, and ME
Perspective		Product and Process
Software support	**Name**	Method Editor
	Type	CAME and MetaCASE
	Use	Research
	Automation level	Medium

Analysis of the Proposal

Since Method Editor is built upon some of the concepts and techniques that were previously introduced by Decamerone, it is also a very complete approach in terms lifecycle coverage and method perspectives. Nonetheless, it also presents some important limitations:

- *Limited process executability*: the main limitation of Method Editor relates to the lack of executability of the UML Activity Diagrams. This limitation prevents the software environments that are obtained by Method Editor from incorporating a process engine, which could offer more advanced execution facilities than the ones that are provided by the navigation browser.
- *Limited tools for product creation*: another important limitation of Method Editor refers to the tools that enable the creation and manipulation of method products during the method execution phase. Specifically, Method Editor is only able to obtain diagram editors, neglecting other types of tools such as code generators, M2M transformations, and textual editors.

Table 3.6 summarizes the relevant information of Method Editor according to the template that is presented in Table 3.1.

3.2.2 Paradigm-based

Paradigm-based Method Engineering is the most general of the three types of Method Engineering. It is based on some initial idea that is expressed as a model (or a metamodel). To construct methods, this model, which is called the *paradigm model*, is evolved into a new model satisfying another engineering objective. The paradigm model thus represents a baseline as-is model that is either instantiated, abstracted, or adapted to obtain to-be models of new methods. It is important to note that, in the case of adaptation, the as-is model and the to-be model are at the same level of abstraction, while in the cases of abstraction and instantiation, these models pertain to different levels [121].

In general, paradigm-based approaches rely on (meta)modeling as their underlying core technique. Since most assembly-based approaches are also based on metamodeling, it is sometimes hard to distinguish paradigm-based approaches from assembly-based approaches. In fact, some approaches can be considered to be both assembly-based and paradigm-based. In this thesis, we consider to be paradigm-based those approaches that (even if they support assembly) do not put especial emphasis on the use of method parts, but rather their focus is on the construction of methods using models and/or metamodels.

The main benefit of paradigm-based Method Engineering approaches is that, since they rely on (meta)modeling as their underlying core technique, they are suitable for defining methods at a high level of abstraction. In the following subsections, we present some of the most relevant paradigm-based approaches.

3.2.2.1 Rolland *et al.* Approach

One of the first paradigm-based Method Engineering approaches was presented by Rolland *et al.* in [142, 143]. This work proposes a decision-oriented process metamodel called NATURE (see Figure 3.11). This metamodel is founded on the idea that software engineers react *contextually* according to the domain knowledge they acquire and react by analogy with previous situations they have been involved in. In order to take into account this basic nature of the development process, NATURE emphasizes the contextual aspect of decisions. Thus, the central concept of the NATURE metamodel is the one of *context*.

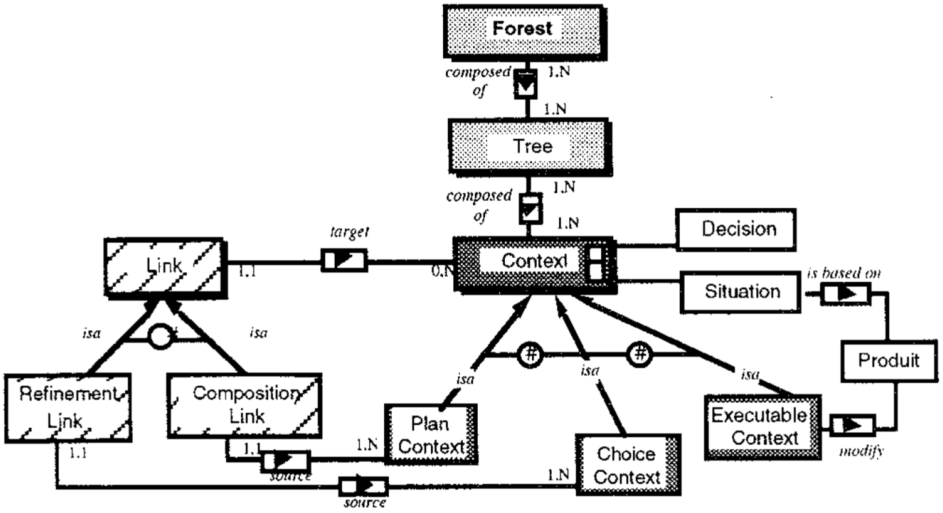


FIGURE 3.11: The NATURE metamodel [143]

In NATURE, contexts are hierarchically grouped into *trees*, which, in turn, are grouped into *forests.* Contexts are tuples <*situation*, *decision*> where *situation* is a part of a product it makes sense to make a decision on (e.g., an entity of an ER model) and *decision* reflects a choice that a software engineer can make at a given moment – e.g., creating (an entity). According to the metamodel, there are three different types of contexts: executable, plan, and choice. *Executable contexts* are the leaves of the trees; that is, the contexts that can be executed by the software engineer. *Plan contexts* represent an abstraction mechanism by which a context (that is viewed as complex) can be decomposed into a number of subcontexts via *composition links.* The execution order of the subcontexts is defined by a dependency graph whose nodes represent contexts and the links define ordered transitions between contexts. On the other hand, *choice contexts* are used to divide contexts into subcontexts by means of *refinement links.* Unlike plan contexts, in choice contexts only one of the subcontexts will be executed. To facilitate the selection of the context to be executed, choice criteria are associated to the contexts. These criteria establish priority rules that allow software engineers to select one alternative among several based on a set of arguments.

It is at this point important to clarify that NATURE looks upon methods from a process-oriented perspective; a complementary product metamodel for NATURE

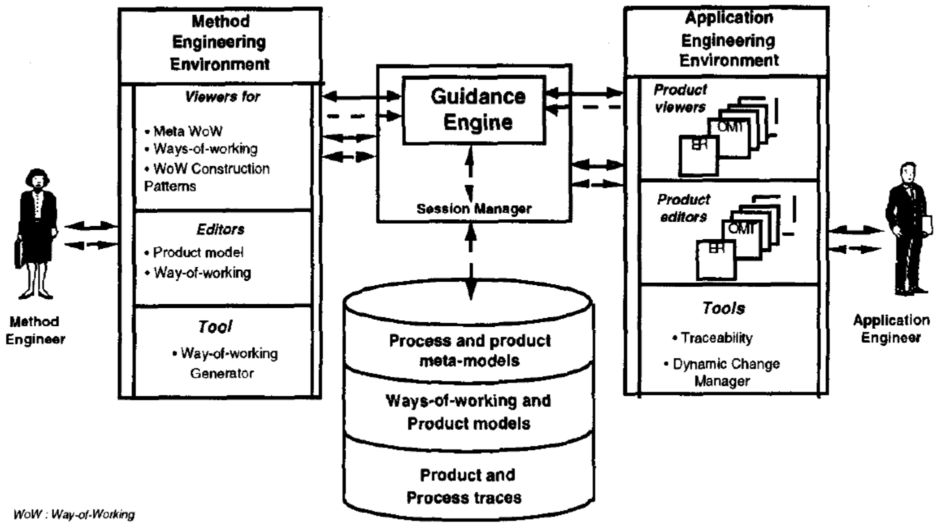


FIGURE 3.12: Architecture of MENTOR [145]

is defined in [144]. In the NATURE formalism, methods are considered in a similar way to Prakash's proposal (see Section 3.2.1.2); that is, methods are viewed as "specification languages" rather than "software development methods". In fact, Prakash emphasizes in [128] that NATURE is complementary to his work, in the sense that NATURE allows method engineers to model the method dynamics; that is, NATURE allows method engineers to model the way of working that drives decision selection, which is external to the method in Prakash's proposal.

With respect to the software support, a software tool supporting NATURE is presented in [145, 146]. This tool is called MENTOR and its overall architecture is graphically depicted in Figure 3.12. As the figure shows, the core component of MENTOR is a *guidance engine*, which provides guidance based on the execution of process models to both software engineers (who construct system specifications) and method engineers (who construct ways of working). In addition to the guidance engine, MENTOR comprises a Method Engineering environment, an Application Engineering environment, and a repository. The *Method Engineering environment* includes viewers, editors, and tools for the method engineer: the product editor and the process editor allow the graphical specification of products and processes, respectively; the method generator aids in the automatic instantiation of predefined generic patterns that are stored in the method base. The

TABLE 3.7: Classification of Rolland *et al.* approach

Rolland *et al.* approach		
Specification language(s)	**Name**	NATURE
	Type	Graphical
	Formality level	High
	Size	Small
Lifecycle coverage		MI
Perspective		Product and Process
Software support	**Name**	MENTOR
	Type	MetaCASE
	Use	Research
	Automation level	Medium

Application Engineering environment constitutes the CASE part of MENTOR, providing tools for supporting the execution of ways of working. Finally, the *repository* stores product and process metamodels; product and process models that are under development; and also process traces resulting from the execution of ways of working.

Analysis of the Proposal

The main strong points of this approach relate to the support for the different method perspectives and also to the software support. Nonetheless, this approach presents one major limitation:

- *Incomplete lifecycle coverage*: in a similar way to all metaCASE environments (such as MERU), MENTOR only supports the technical aspects of methods, even though methods comprise other aspects, such as the activity-related [21]. Thus, MENTOR does not allow method engineers to specify method elements such as tasks, activities, roles, documents, or quality metrics, which we consider to be mandatory in any approach that supports the design and execution of software development methods. For this reason, we consider MENTOR to be a metaCASE environment that focuses exclusively on the method implementation phase of the Method Engineering lifecycle, rather than a CAME environment that supports other phases, such as the method design and the method execution.

Table 3.7 summarizes the relevant information of the approach that is proposed by Rolland *et al.* according to the template that is presented in Table 3.1.

3.2.2.2 MetaEdit+

MetaEdit+ [40, 42, 147, 148] is a commercial metaCASE environment that allows users to build graphical DSLs (which are also called "methods" in MetaEdit+) and code generators that are adapted to specific application domains. The implementation of generators is performed by means of MERL, a textual DSL that is designed for turning models into text. The construction of methods is performed at a higher level of abstraction without having to write source code. For the construction of methods, MetaEdit+ provides a (meta)metamodeling language called GOPRR and a tool suite for defining method concepts, their properties, associated rules, symbols, and checking reports. When a method is defined, MetaEdit+ automatically provides CASE tool functionality for using the method: diagramming editors, browsers, etc.

Figure 3.13 shows the (meta)metamodel of GOPRR. The name GOPRR is an acronym that stands for the metatypes that are supported by the language: Graph, Object, Property, Role, and Relationship. An additional metatype (Port) was added afterwards, and, therefore, the language was renamed GOPPRR. The semantics of all of these metatypes is the following. A *Graph* specifies one modeling language or method (e.g., the ER notation or BPMN). *Objects* are the basic concepts of the methods (e.g., the entities of the ER notation or the tasks of BPMN); objects typically appear as shapes in the diagrams. *Properties* are characterizing attributes that are attached to the objects; properties typically appear as textual labels or icons in the diagrams (e.g., the names of the entities of the ER notation or the markers of the tasks of BPMN). *Relationships* are the concepts that connect the objects; typically, relationships are represented as lines. *Roles* define the ways in which objects participate in relationships; roles typically appear as end points of relationships (e.g., as arrowheads). Finally, *ports* allow method engineers to specify additional semantics with respect to how roles can be connected with objects; typically, in the diagrams, ports are attached to the objects in order to restrict the possible points of connection between the objects and the roles.

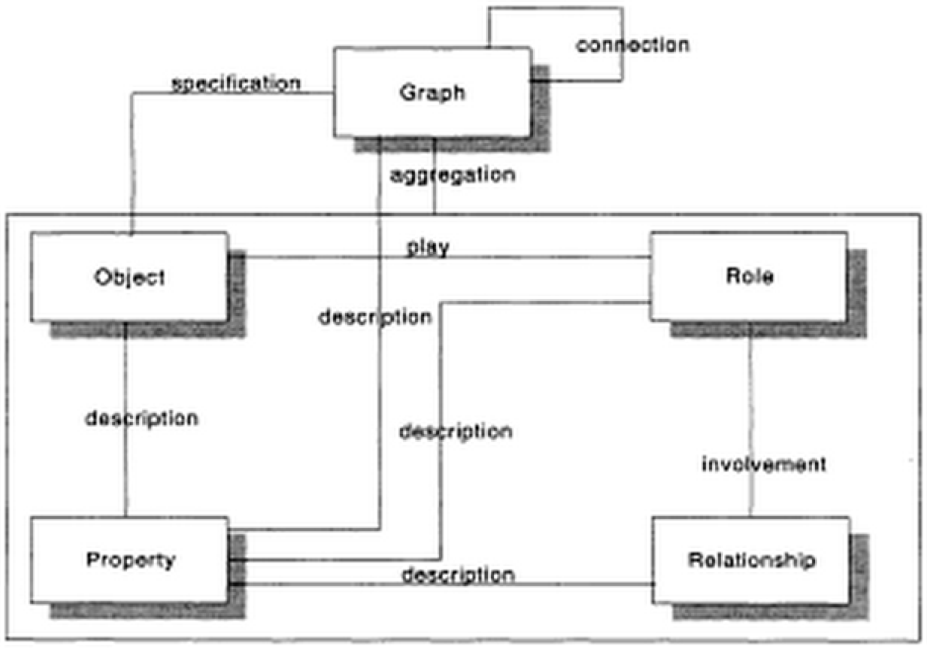


FIGURE 3.13: The GOPRR metamodel [149]

Since the GOPRR language is exclusively product-oriented, an extension was proposed in [150] to provide process support in MetaEdit+. The process-oriented extension of GOPRR is called GOPRR-p. The main goal of GOPRR-p is to enable the definition of process models that provide automated guidance for software engineers when they use the environments that are generated with MetaEdit+. To achieve this goal, GOPRR-p extends GOPRR with two additional metatypes: Process Element and Action. *Process Elements*, which inherit from the GOPRR object metatype, represent the basic concepts of the process models, such as the tasks or the deliverables. To specify the semantics of process elements (i.e., how these elements must be supported in a CASE environment), actions can be attached to them. *Actions* capture and conceptualize CASE environment behavior; for instance, opening an editor or closing dialog. Thus, when we associate actions with process elements, the process elements become objects that execute actions within the context of the CASE environment.

The general architecture of MetaEdit+ is illustrated in Figure 3.14. The heart of

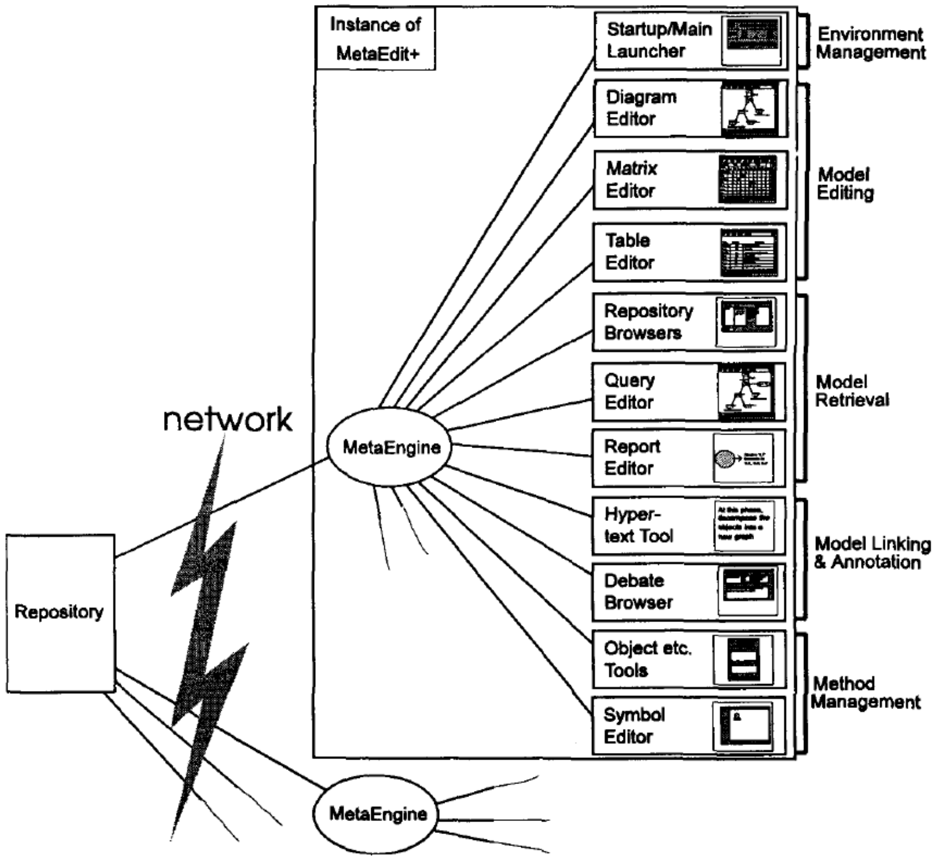


FIGURE 3.14: MetaEdit+ architecture [40]

the environment is the MetaEngine, which implements the underlying conceptual data model (i.e., GOPPRR). Tools within MetaEdit+ request services to the MetaEngine in order to access and manipulate repository data. These tools are classified into five distinct families: environment management tools (for managing features of the environment and its main components); model editing tools (for creating, modifying, and deleting model instances or their parts); model retrieval tools (for retrieving design objects and their instances from the repository for reuse and review); model linking and annotation tools (for linking design objects for traceability and memorization); and method management tools (for method specification, management, and retrieval).

TABLE 3.8: Classification of MetaEdit+

MetaEdit+			
Specification language(s)	**Name**	MERL	GOPPRR
	Type	Textual	Graphical
	Formality level	Medium	Low
	Size	Big	Small
Lifecycle coverage		MI	
Perspective		Product	
Software support	**Name**	MetaEdit+	
	Type	MetaCASE	
	Use	Research and Commercial	
	Automation level	High	

Of especial relevance for this thesis are the method management tools. The primary goal of this tool family is to enable flexibility and ease in the construction and use of methods. The method management tools family consists of the following main parts: a method base (which stores method fragments and symbols used for representing object types); a method assembly system (which consists of specialized tools for method assembly, such as metamodel editors and a symbol editor); and the environment generation system (which consists of generators that take as input method specifications and deliver usable and user-friendly software environments).

Analysis of the Proposal

The MetaEdit+ environment represents an effective solution for defining modeling languages and code generators. It is easy to use, well-documented, and it also has an intuitive graphical design. Furthermore, to the best of our knowledge, MetaEdit+ is the only Method Engineering tool that has been commercialized. However, it presents two important limitations:

- *Incomplete lifecycle coverage*: MetaEdit+ falls short in providing support for the design of software development methods since it only focuses on the technical aspects of Method Engineering; for instance, MetaEdit+ allows method engineers to develop a code generator that automates a method task, but it does not support the creation of a method model that provides a conceptual definition for this task.

- *Fully product-oriented*: in a similar way to MERU, MetaEdit+ is fully product-oriented. This is because the process-oriented extension of its conceptual data model (i.e., GOPRR-p) is not implemented in the commercial version of the tool.

Table 3.8 summarizes the relevant information of MetaEdit+ according to the template that is presented in Table 3.1.

3.2.2.3 Ralyté *et al.* Approach

Ralyté *et al.* present in [151] an approach for Method Engineering that is based on the evolution of an existing method, model, or metamodel into a new one better adapted for a given situation and/or satisfying a different engineering objective. In a similar way to their assembly-based approach (see Section 3.2.1.3), Ralyté *et al.* propose a **generic process model** for guiding method engineers during the evolution-driven process. The process model proposes several strategies to evolve the initial paradigm model into a new one and provides guidelines supporting these strategies.

Figure 3.15 graphically depicts the process model in the form of a map [132]. As the figure shows, Ralyté *et al.* suggest to start the evolution-driven process by constructing the product part of the method. This is indicated by the *construct a product model* intention. To achieve this intention, four different strategies can be followed: abstraction, instantiation, utilization, or adaptation. The *abstraction strategy* is followed to obtain the product model by raising (or lowering) the abstraction level of a given model. The *instantiation strategy* is followed to obtain the product model by instantiating a metamodel. The *utilization strategy* and the *adaptation strategy* assist, respectively, in the adjustment of a model or a metamodel to some specific circumstances.

When the product model is built, the method engineer can proceed to the next intention: construct a process model. The reason why the *construct a process model* intention follows the *construct a product model* intention (and not the opposite) is because process models must conform to the product model (i.e., process steps, activities, actions must refer to product model parts in order to construct, refine, or transform them). To achieve the *construct a process model*

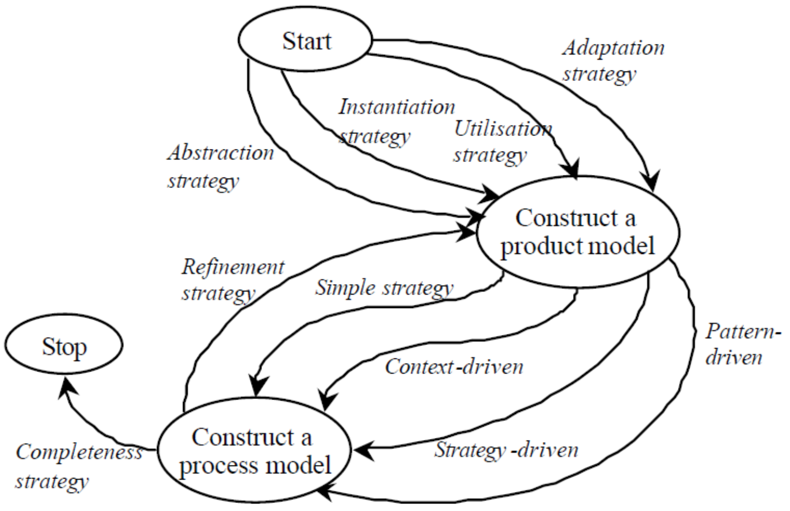


FIGURE 3.15: Process model of the paradigm-based approach [151]

intention, four different strategies can be applied: simple, context-driven, pattern-driven, and strategy-driven. The *simple strategy* is useful to define simple process models that can be expressed informally, for example, as textual descriptions. The *context-driven* strategy suggests the use of the NATURE process modeling formalism (which is summarized in Section 3.2.2.1). The *pattern-driven* strategy suggests the use of a catalogue of patterns. Each pattern identifies a problem that can occur during the construction of the product model and proposes a solution applicable every time that the problem appears. The *strategy-driven* strategy allows method engineers to combine several process models (i.e., maps) into one complex process model.

Finally, note that, during the *construct a process model* intention, backtracking to the *construct a product model* intention is always possible thanks to the *refinement strategy*. This strategy allows method engineers to improve the product model based on data obtained during the construction of the process model. Additionally, to check whether the final models match the initial requirements, method engineers shall apply the *completeness strategy*. If the response is positive, then the evolution-driven process ends. In any other case, either the product model, the process model, or both have to be revised to gain the required method completeness.

TABLE 3.9: Classification of Ralyté *et al.* approach (paradigm-based)

Ralyté *et al.* approach (paradigm-based)				
Specification language(s)	**Name**	UML	Map	NATURE
	Type	Graphical	Graphical	Graphical
	Formality level	Medium	High	High
	Size	Big	Small	Small
Lifecycle coverage		MD		
Perspective		Product and Process		
Software support	**Name**	x		
	Type	x		
	Use	x		
	Automation level	x		

Analysis of the Proposal

Table 3.9 summarizes the relevant information of the paradigm-based approach that is proposed by Ralyté *et al.* according to the template that is presented in Table 3.1. The main limitations of this approach are the following:

- *Lack of software support*: the proposal of Ralyté *et al.* has neither been implemented as a CAME environment nor as a metaCASE tool. Software support is necessary in order to enable the practical application of the approach.

- *Incomplete lifecycle coverage*: in a similar way to most Method Engineering approaches, the approach of Ralyté *et al.* does not address the method implementation and the method execution phases of the Method Engineering lifecycle.

- *Limited process support*: another limitation concerns the process support since this approach only considers the small-grained processes that specify how to develop particular product models, thereby neglecting the overall process of software development.

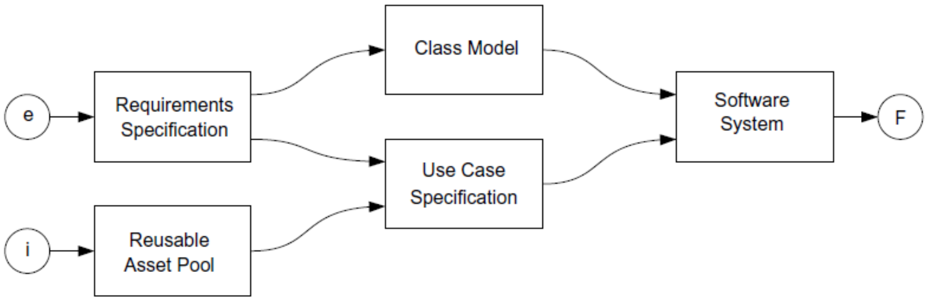
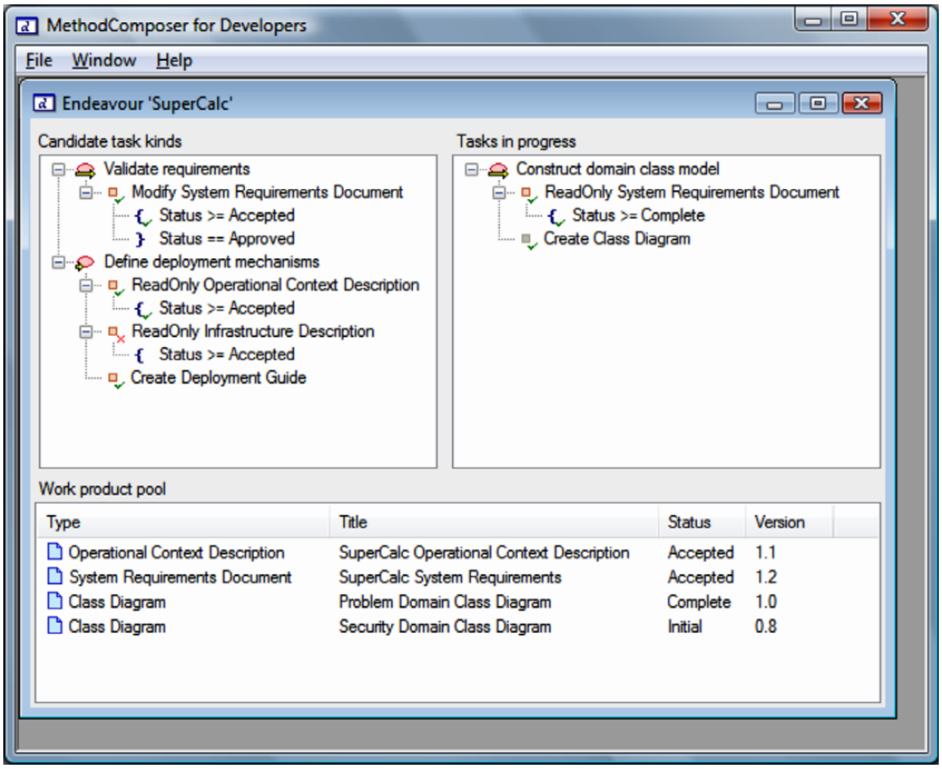


FIGURE 3.16: Example of product-oriented method [22]

3.2.2.4 Work Product Pool Approach

Gonzalez-Perez *et al.* introduce in [22, 152] the work product pool approach. This approach is founded on the idea that traditional process-oriented methods are too prescriptive and rigid, and most of them end up being ignored or bypassed. The authors argue that, since the ultimate aim of software development is to deliver a software product, methods should be described in terms of the intermediate products that are necessary to reach this software product (plus the processes that are required to produce the intermediate products). To obtain such product-oriented methods, the authors suggest that method engineers start formulating the following question: "what products are necessary to easily and directly obtain the final software product?". This question is done recursively; thus, only the products that are vital to the construction of the final software product are identified. When these products are identified, the processes for creating them can be defined. For the definition of methods following the work product pool approach, Gonzalez-Perez *et al.* adopt the ISO/IEC 24744 standard.

Figure 3.16 illustrates a simple example of a product-oriented method. The ultimate objective of this method is a finished *software system*. This product is represented in the diagram as the right-most box. The arrows indicate that, to obtain the final software system, it is necessary to develop a *class model* and a *use case specification*. These products, in turn, require the development of a *requirements specification* and a *reusable asset pool*. The letter "F" in a small circle means final fulfillment. On the other hand, the letter "e" means that the target



Candidate task kinds:

- Validate requirements
  - Modify System Requirements Document
    - Status >= Accepted
    - Status == Approved
- Define deployment mechanisms
  - ReadOnly Operational Context Description
    - Status >= Accepted
  - ReadOnly Infrastructure Description
    - Status >= Accepted
  - Create Deployment Guide

Tasks in progress:

- Construct domain class model
  - ReadOnly System Requirements Document
    - Status >= Complete
  - Create Class Diagram

Work product pool:

Type	Title	Status	Version
Operational Context Description	SuperCalc Operational Context Description	Accepted	1.1
System Requirements Document	SuperCalc System Requirements	Accepted	1.2
Class Diagram	Problem Domain Class Diagram	Complete	1.0
Class Diagram	Security Domain Class Diagram	Initial	0.8

FIGURE 3.17: Screenshot of Method Composer [22]

product is externally provided, while the letter "i" means that the target product is internally available.

The execution of this kind of methods relies on the existence of a **work product pool**. This pool is a central repository that stores all the work products that exist at any point in time during a method execution. Initially, when the execution starts, the work product pool contains few products; specifically, it contains the products that are internally available and also those that are externally provided. As tasks begin to be carried out, existing work products are read, modified, and deleted, and new work products are created, changing the population of the pool. Eventually, the final system appears in the pool and the method execution can be considered to be complete. In order to enable this product-oriented method execution, Gonzalez-Perez *et al.* [22] define a set of **algorithms** that calculate, based on the content of the work product pool, the tasks to be performed by software engineers. This contextual guidance contrasts with the guidance that is provided by process-oriented methods, which suggest the tasks to be performed based on a pre-defined and fixed plan.

The work product pool approach is implemented in a CAME environment that is called Method Composer [22, 153]. This CAME environment allows method engineers to define product-oriented methods following the ISO/IEC 24744 standard and it also allows software engineers to execute these methods. For the execution of methods, Method Composer implements the algorithms that are proposed by the work product pool approach. These algorithms allow software engineers to display a list of candidate tasks at any point in time and choose, from that list, the task that they wish to execute. The list of candidate tasks is dynamically calculated on the fly according to the role of the software engineer, the organization's context, and, most importantly, the content of the work product pool. Figure 3.17 illustrates how Method Composer displays the list of candidate tasks as well as the tasks that are in progress. The work product pool pane appears at the bottom, showing an individual work product per row. Users' actions of any type (e.g., creation of a work product, modification of an existing one, or completion of a task) must be reported manually to the system (even if they do not alter the work product pool) so that Method Composer updates the state of the method instance that is being executed.

Analysis of the Proposal

Table 3.10 summarizes the relevant information of the Work Product Pool approach according to the template that is presented in Table 3.1. We consider that the main limitations of this approach are the following:

- *Lack of support for method implementation*: the major limitation of this approach is the lack of support for the method implementation phase of the Method Engineering lifecycle. While Method Composer does provide support for method execution, this support only involves guidance on what tasks to execute. The CAME environment does not incorporate tools that allow software engineers to perform the tasks and develop the work products (e.g., textual/graphical editors, M2M transformations, and code generators). This deficiency restricts the level of automation that Method Composer can achieve since all of the software engineers' actions occur outside

Table 3.10: Classification of the Work Product Pool approach

Work Product Pool approach		
Specification language(s)	**Name**	ISO/IEC 24744
	Type	Graphical
	Formality level	Medium
	Size	Big
Lifecycle coverage		MD and ME
Perspective		Product
Software support	**Name**	Method Composer
	Type	CAME
	Use	Research
	Automation level	Low

of the context of the CAME environment. For instance, if a software engineer transforms (either automatically or manually) a UML class model into an ER model, he/she is forced to manually notify the system about this action. In an ideal scenario, Method Composer would support the M2M transformation, and, thus, it could automatically transform the models as well as automatically set the corresponding method task as executed if the transformation finished successfully.

- *Fully product-oriented*: this approach is based on a work product pool that provides a fully product-oriented view of the development process.

3.2.3 Configuration-based

Method engineers in practice often take an existing organization-wide method (a.k.a. *base method*) as their point of departure, rather than a set of method parts for assembly [26]. Based on this observation, configuration-based Method Engineering aims to facilitate the adaptation of this base method to the actual needs of the development project at hand. This adaptation to situational factors is collectively known as "method configuration".

In a similar way to paradigm-based approaches, configuration-based approaches take a model as starting point and evolve this model into a new one that satisfies another engineering objective. Therefore, it is sometimes difficult to distinguish configuration-based approaches from paradigm-based approaches. In fact, some approaches can be considered to be both configuration-based and paradigm-based. In this thesis, we consider that paradigm-based approaches are more general, in the sense that any approach that supports the construction of methods using models and/or metamodels can be considered to be paradigm-based. In contrast, configuration-based approaches focus on (semi)automating the process of adaptation of an existing base method to various situational factors or project characteristics.

The use of configuration-based Method Engineering provides three major benefits. First, it reduces the workload of method engineers, since project-specific methods are obtained (semi)automatically, rather than manually by assembly, instantiation, or abstraction. Second, by adopting a configuration-based approach, organizations gain the required flexibility to build methods that are adapted to the context of use. Finally, since project-specific methods are defined in-house, software engineers are motivated to use them due to the feeling of method ownership [36]. In the following subsections, we present some of the most relevant configuration-based approaches.

3.2.3.1 Deneckère *et al.* Approach

Deneckère *et al.* propose in [154, 155] a Method Engineering approach for enhancing existing methods with new features and properties. These kinds of enhancements represent a type of method configuration that is known as extension-based Method Engineering [9]. One example of method extension is introducing temporal features so that the method deals more systematically with the representation of calendar time.

In order to carry out the extension of methods, Deneckère *et al.* advocate the use of patterns. The general idea concerning any type of pattern is that a pattern relates a recurrent *problem* to a *solution*, which can be applied any time the problem occurs [154]. Deneckère *et al.* adopt this notion of pattern and apply it in context of extension-based Method Engineering; the result is the notion of extension pattern. According to Deneckère *et al.*, an **extension pattern** is a reusable component that helps method engineers to identify typical extension

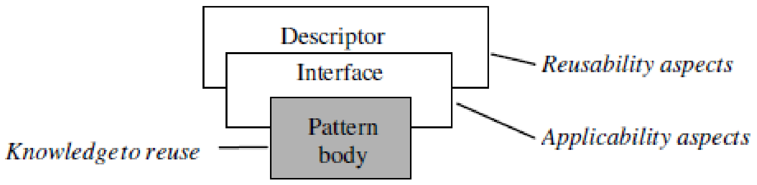


FIGURE 3.18: Structure of an extension pattern [154]

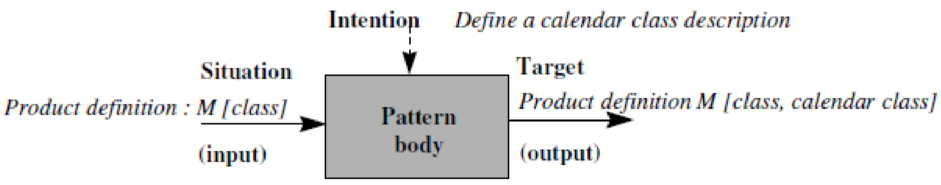

Intention
Define a calendar class description
Situation
Product definition : M [class]
(input)
Pattern body
Target
Product definition M [class, calendar class]
(output)

FIGURE 3.19: Interface of an extension pattern [154]

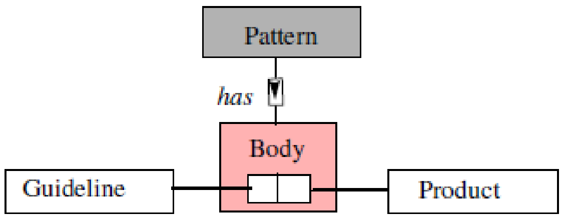


FIGURE 3.20: Body of an extension pattern [154]

situations (i.e., the recurrent problem) and provides guidance (i.e., the solution) on how to perform the required extensions [9].

Figure 3.18 shows the general structure of an extension pattern. As the figure shows, an extension pattern comprises three main parts: descriptor, interface, and body. The *descriptor* manages the reusability aspects; that is, the context in which it is useful to reuse the pattern. The *interface* (see Figure 3.19) manages the applicability aspects; that is, in which situation and for which intention the pattern is applicable. The interface is a triplet <*situation, intention, target*> where *situation* is a (product) part of the method to be extended, *intention* refers to the extension to apply, and *target* is the output of this extension. Finally, the *body* of an extension pattern (see Figure 3.20) manages the reusable knowledge; that is, the solution proposed by the pattern. The body contains the guidelines

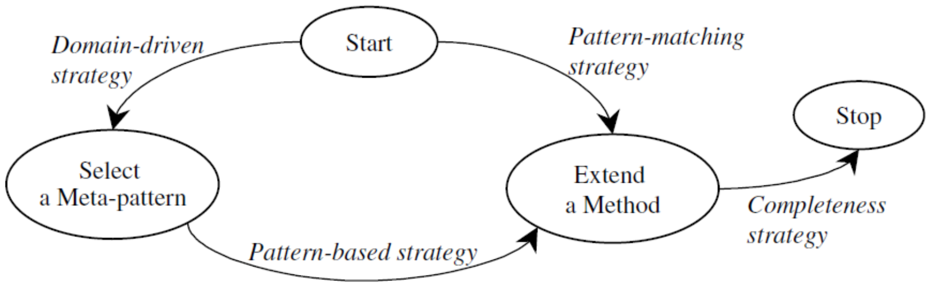


FIGURE 3.21: Process model for method extension [9]

to be followed when the pattern is applied (i.e., the process part of the pattern) as well as the definition of the product under modification. The product part of a pattern typically represents the concepts of a model to extend. On the other hand, the guidelines can be expressed either formal or informally. Formal guidelines are expressed using the NATURE process modeling formalism [142], while informal guidelines are expressed in natural language. For the definition of the product part of a pattern, Deneckère *et al.* make use of the class diagrams of the UML standard.

In order to facilitate the process of method extension, Deneckère *et al.* propose a **generic process model** for guiding method engineers. Figure 3.21 graphically depicts this process model in the form of a map. As the figure shows, Deneckère *et al.* advocate two different ways to extend a method: (1) directly through the *pattern-matching strategy* or (2) through the path *select a meta-pattern*, *extend a method* applying the *pattern-based strategy.* The former helps to match extension patterns stored in a library to the requirements of the extension; the latter selects first a meta-pattern corresponding to the extension domain, and, then, guides the method extension by applying the patterns suggested by the meta-pattern. Both approaches use a library of extension patterns, but they do it in different ways. The domain-centric way exploits the fact that a set of patterns can be embodied in a meta-pattern that is suitable for a specific domain (e.g., temporal data structures). If the required extension does not correspond to a well-identified extension domain, then the pattern-matching approach must be selected. In [9], Ralyté *et al.* provide a more in-depth summary about each of these two approaches: domain-driven and pattern-matching. The notion of meta-pattern is further discussed by Deneckère in [156].

Table 3.11: Classification of Deneckère *et al.* approach

Deneckère *et al.* approach			
Specification language(s)	**Name**	UML	NATURE
	Type	Graphical	Graphical
	Formality level	Medium	High
	Size	Big	Small
Lifecycle coverage		MD	
Perspective		Product and Process	
Software support	**Name**	x	
	Type	x	
	Use	x	
	Automation level	x	

Analysis of the Proposal

Table 3.11 summarizes the relevant information of the extension-based approach that is proposed by Deneckère *et al.* according to the template that is presented in Table 3.1. The limitations of this approach are similar to those that were identified for Ralyté's approach in Sections 3.2.1.3 and 3.2.2.3 These limitations are the following:

- *Lack of software support*: the approach of Deneckère *et al.* lacks a software tool that enables the practical application of extension-based Method Engineering.

- *Incomplete lifecycle coverage*: the approach of Deneckère *et al.* focuses exclusively on the method design phase of the Method Engineering lifecycle. Thus, it enables the conceptual specification of methods, but neglects their technical support (i.e., the construction of integrated environments that support the method execution).

- *Limited process support*: another limitation of this approach concerns the support for the process part of methods. Similarly to the method chunks that are proposed by Ralyté *et al.* (see Section 3.2.1.3), the extension patterns combine a product part with a process part. Thus, they define small-grained processes that establish how to manipulate particular product parts, but they neglect the overall process of software development.

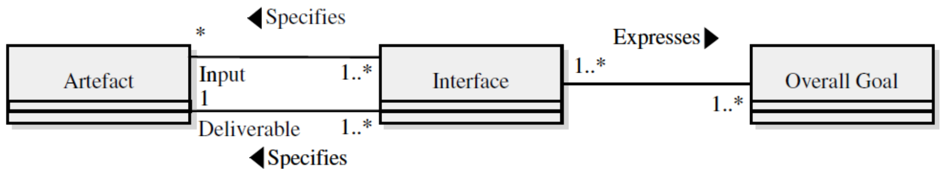


FIGURE 3.22: The method component concept (external view) [158]

3.2.3.2 Method for Method Configuration

One of the most relevant configuration-based Method Engineering approaches was presented by Karlsson *et al.* in [4, 157]. This work defines a meta-method that is called Method for Method Configuration (MMC). In the MMC, software development methods are viewed as collections of reusable parts that are called method components.

A **method component** can be defined as "a self-contained part of a software development method expressing the transformation of one or several artifacts into a defined target artifact and the rationale for such a transformation" [33, 158]. Similarly to method chunks [130], method components subscribe to the idea of information hiding [26], inspired by the traditional use of the component concept in software engineering [159]. For this reason, the method component construct consists of two basic views: the external view and the internal view, which are shown in Figures 3.22 and 3.23, respectively. The external view is the one that enables information hiding and reduces the focus on details during method configuration. This view is given by an interface that defines the recommended input *artifacts* of the component, the component's *deliverable*, and the component's *overall goals*. On the other hand, the internal view consists of *method elements* (which can be thought of as method fragments at lower layers of granularity than the component itself), and *goals* (which are anchored in the *values* of the method creator [33]). All of the method elements in a method component contribute to achieve its overall goals. The most important specialization of the method element concept is the *artifact* class, since method component is an artifact-centric concept. The input artifacts of the component are transformed into a deliverable by means of *actions* that are performed by *actor roles*. When an actor performs an action, *concepts* and *notations* are used to manipulate the artifacts.

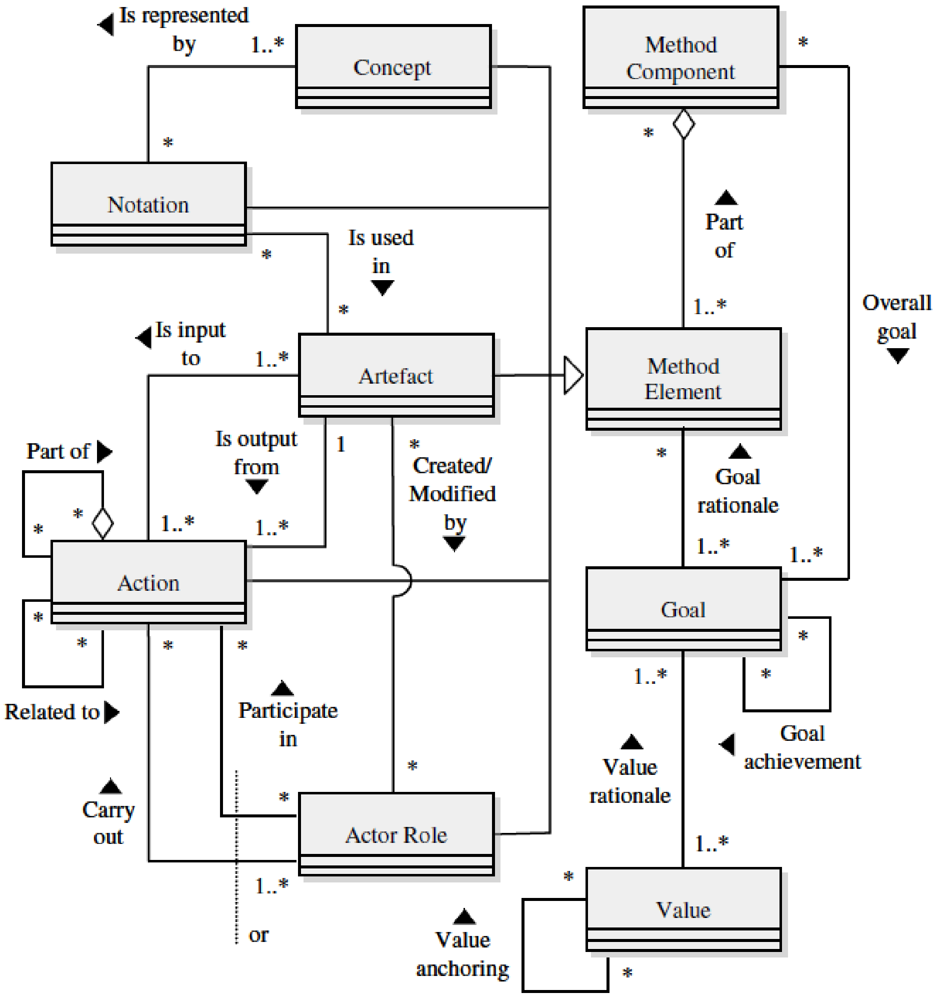


FIGURE 3.23: The method component concept (internal view) [158]

In contrast to assembly-based approaches, whose point of departure is a set a method parts for assembly, the MMC takes a base method as starting point. To perform the configuration of the base method, the MMC relies on a set of concepts, which are shown in Figure 3.24. Aside from the method component concept, which is explained above, there are two additional constructs that represent the key drivers of the process of method configuration: configuration package and configuration template. Prior to the definition of these two concepts, Karlsson *et al.* define the concept of project *characteristic* as "a delimited part of a *development*

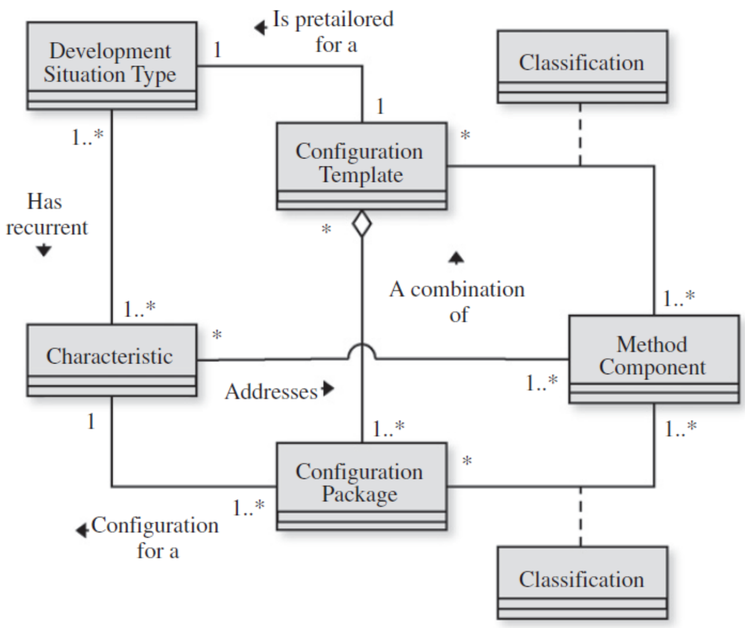
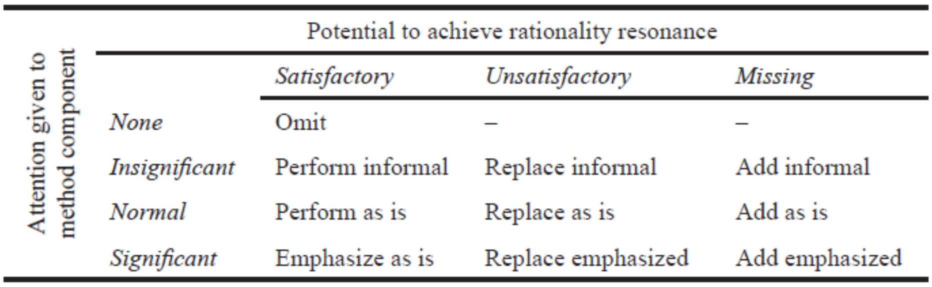


FIGURE 3.24: Main concepts of MMC [39]

situation, focusing on a certain problem or aspect which the method configuration aims to solve or handle". One example of project characteristic is "Data persistence", which has two possible values: "yes" and "no". If we consider a project that is characterized as "Data persistence = no", then the project needs a configuration of the base method that does not include, e.g., method components that deal with database design. A configuration package encapsulates such a method configuration. Karlsson *et al.* define **configuration package** as "a configuration of the base method that is suitable for one specific project characteristic's value". Within a configuration package, the components of the base method are classified according to the classification schema[4] that is shown in Table 3.12. When a method engineer classifies a method component, he/she is specifying how the component will be executed in the projects that exhibit the characteristic's value that the configuration package represents. For instance, in a configuration package that represents the value "Data persistence = no", the method components dealing with database design will be classified as "Omit". Another task of method engineers (aside from classifying method components) is to combine the resulting

[4]For an in-depth description of the classification schema, see [157]



Attention given to method component	Potential to achieve rationality resonance		
	Satisfactory	*Unsatisfactory*	*Missing*
None	Omit	–	–
Insignificant	Perform informal	Replace informal	Add informal
Normal	Perform as is	Replace as is	Add as is
Significant	Emphasize as is	Replace emphasized	Add emphasized

TABLE 3.12: Classification schema for method components [157]

configuration packages to obtain larger structures. This is because, in the real world, development projects are not simple enough to be captured through one single characteristic. These larger structures are called configuration templates. A **configuration template** can be defined as "a combined method configuration, based on configuration packages, for a set of recurrent project characteristics" [4]. Configuration templates can therefore be seen as project-specific methods that are fine-tuned and adapted to the various characteristics of particular development projects.

Based on the concepts that are shown in Figure 3.24, Karlsson *et al.* define the MMC. The activities and artifacts that comprise the MMC are graphically depicted in Figure 3.25. As the figure shows, the MMC is divided into three sections. Section 1 deals with the construction of configuration packages. Specifically, the activity "*Identify Development Situations and Characteristics*" indicates that, first, development projects must be abstracted as sets of key-distinctive characteristics. Then, these characteristics are stored in the "*Base of Development Situations and Characteristics*" so that they can be later used to build configuration packages[5]. This is done in the activity "*Administration of Configuration Packages*". The resulting packages are stored in the "*Base of Configuration Packages*". Section 2 of the MMC deals with configuration templates. Specifically, this section involves two main activities: "*Selecting Configuration Packages*" and "*Combining Configuration Packages*". The former is carried out matching project characteristics with existing configuration packages. The latter involves the resolution of classification conflicts between configuration packages and the evaluation of the consistency of the resulting configuration templates. These templates are stored

[5]Note that choosing a base method is outside the scope of the MMC.

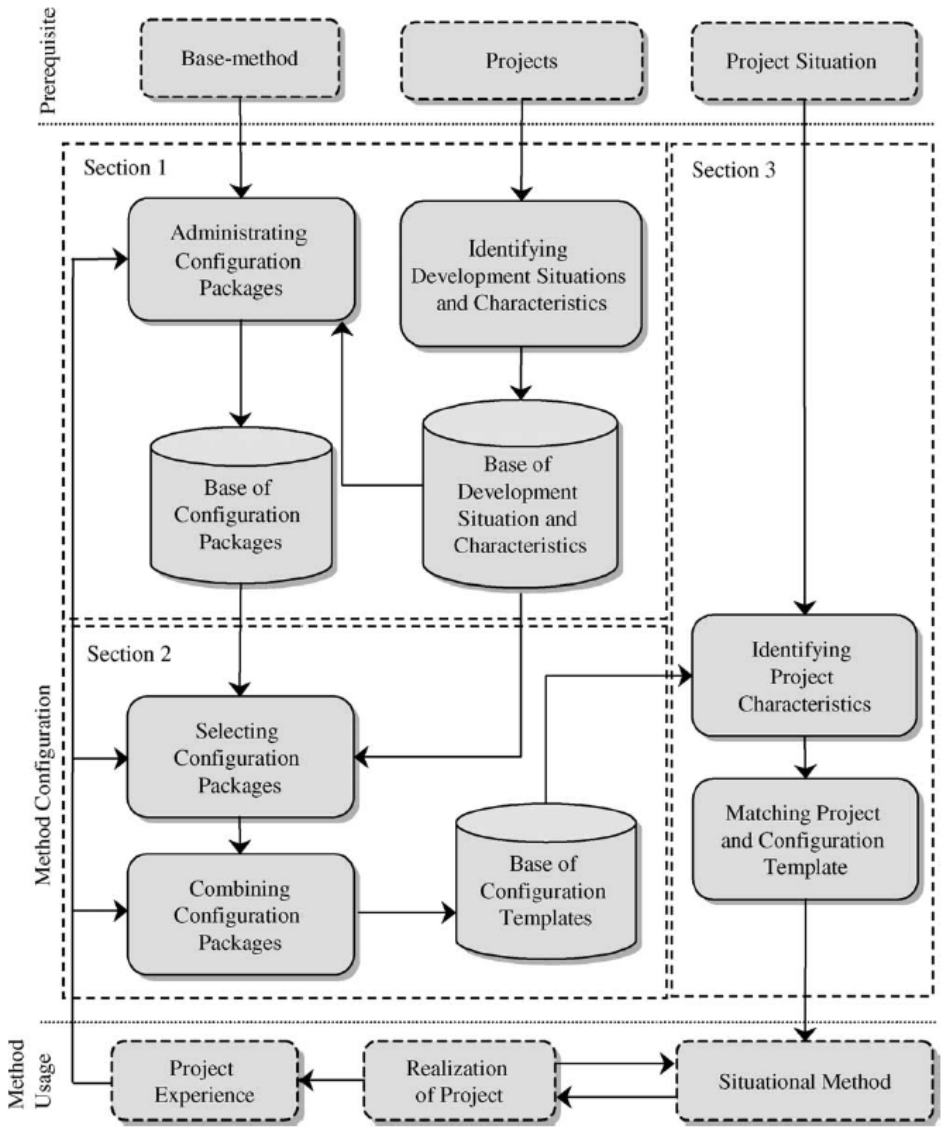
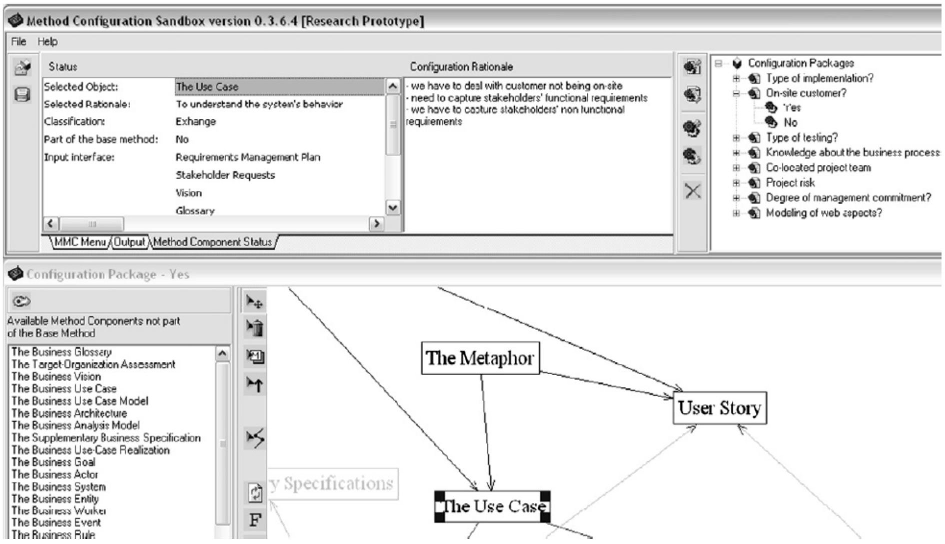

Prerequisite
Base-method
Projects
Project Situation
Section 1
Section 3
Administrating Configuration Packages
Identifying Development Situations and Characteristics
Base of Configuration Packages
Base of Development Situation and Characteristics
Section 2
Identifying Project Characteristics
Selecting Configuration Packages
Matching Project and Configuration Template
Combining Configuration Packages
Base of Configuration Templates
Method Configuration
Method Usage
Project Experience
Realization of Project
Situational Method

FIGURE 3.25: The Method for Method Configuration [4]


Method Configuration Sandbox version 0.3.6.4 [Research Prototype]

File Help

Status

Selected Object:	The Use Case
Selected Rationale:	To understand the system's behavior
Classification:	Exhange
Part of the base method:	No
Input interface:	Requirements Management Plan
	Stakeholder Requests
	Vision
	Glossary

Configuration Rationale

- we have to deal with customer not being on-site
- need to capture stakeholders' functional requirements
- we have to capture stakeholders' non functional requirements

Configuration Packages
- Type of implemenlation?
- On-site customer?
  - Yes
  - No
- Type of testing?
- Knowledge about the business process
- Co-located project team
- Project risk
- Degree of management commitment?
- Modeling of web aspects?

Configuration Package - Yes

Available Method Components not part of the Base Method

The Business Glossary
The Target-Organization Assessment
The Business Vision
The Business Use Case
The Business Use Case Model
The Business Architecture
The Business Analysis Model
The Supplementary Business Specification
The Business Use-Case Realization
The Business Goal
The Business Actor
The Business System
The Business Entity
The Business Worker
The Business Event
The Business Rule

FIGURE 3.26: Screenshot of MC Sandbox [39]

in the "*Base of Configuration Templates*". Finally, section 3 deals with starting a specific project. This involves an investigation of the project situation, as indicated by the activity "*Identifying Project Characteristics*". If a matching configuration template is found, it must be fine-tuned into a situational method. This is illustrated by "*Matching Project and Configuration Template*". The resulting situational method is used during the project. The feedback that is obtained is used for improving the existing configuration packages and configuration templates.

The MMC is implemented in a CAME environment that is called MC Sandbox [39, 160]. This CAME environment supports five basic operations: definition of method components, edition of the base method, definition of configuration packages, definition of configuration templates, and definition of situational methods. Figure 3.26 shows a screenshot of MC Sandbox. As the figure shows, the screen of MC Sandbox is divided horizontally. The lower section contains the method modeling area, which makes use of the external view of method components. These components are depicted as rectangles; the arrows represent flow of artifacts. When a method component is selected in the modeling view, the content of the component interface is presented in the upper left section of the screen. The right part of the upper section shows as a tree structure the complete set

Table 3.13: Classification of the Method for Method Configuration

Method for Method Configuration		
Specification language(s)	**Name**	x
	Type	x
	Formality level	x
	Size	x
Lifecycle coverage		MD
Perspective		Product
Software support	**Name**	MC Sandbox
	Type	CAME
	Use	Research
	Automation level	High

of existing configuration packages. This tree structure is sorted by the project characteristics that the configuration packages belong to.

Analysis of the Proposal

One important advantage of the MMC is the high level of automation that is achieved during the configuration process: methods are (semi)automatically obtained from the method requirements; that is, methods are obtained directly from the characteristics of the projects where the methods will be applied. This reduces the complexity of the Method Engineering approach, and, as a consequence, facilitates its application in practice. In contrast to this advantage, we also identified two important limitations:

- *Incomplete lifecycle coverage*: the MMC is exclusively focused on the conceptual configuration of methods so that they meet particular project needs; thus, it only supports the method design phase of the Method Engineering lifecycle.

- Fully product-oriented: the MMC does not support the process part of methods since the method component construct is exclusively product-oriented.

Table 3.13 summarizes the relevant information of the MMC according to the template that is presented in Table 3.1.

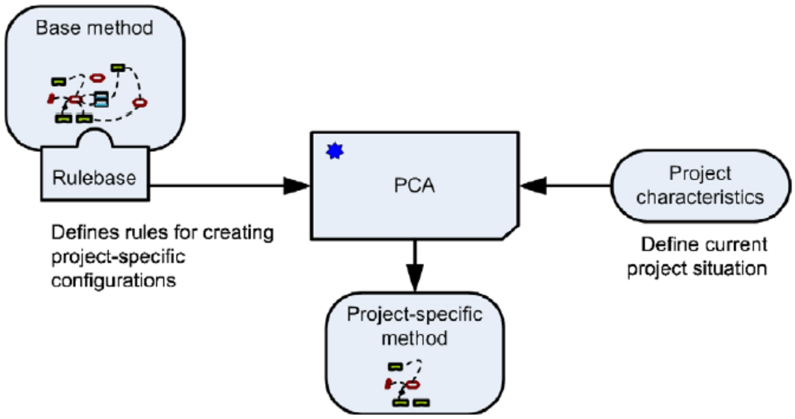

Base method
Rulebase
Defines rules for creating project-specific configurations
PCA
Project characteristics
Define current project situation
Project-specific method

FIGURE 3.27: Overview of the PCA [6]

3.2.3.3 Process Configuration Approach

Bajec *et al.* propose in [6] the Process Configuration Approach (PCA). The general idea that lies behind the PCA is illustrated in Figure 3.27. In the PCA, project-specific methods are created by selecting components from a base method. The selection of components is done automatically by processing rules that tell in what circumstances or project situations it is compulsory, advisable, or discouraged to use a particular component.

Unlike the MMC, the PCA does not define its own notion of method component but rather defines a **generic data structure** underpinning any arbitrary method (i.e., a metametamodel). This generic data structure makes the PCA neither a product-oriented nor a process-oriented approach; instead, the PCA is a generic approach that could be applied to any model that instantiates the metametamodel.

Figure 3.28 illustrates the main concepts of the generic data structure. As the figure shows, the classes of the metametamodel are *MetaElement* (which can be content elements such as activities, tools, and roles; or process flow elements, such as decision nodes, joins, and synchronisations) and *MetaLink* (which represents links between metaelements). By using such a generic data structure, a base method is represented as a structure of instances of metaelements and metalinks,

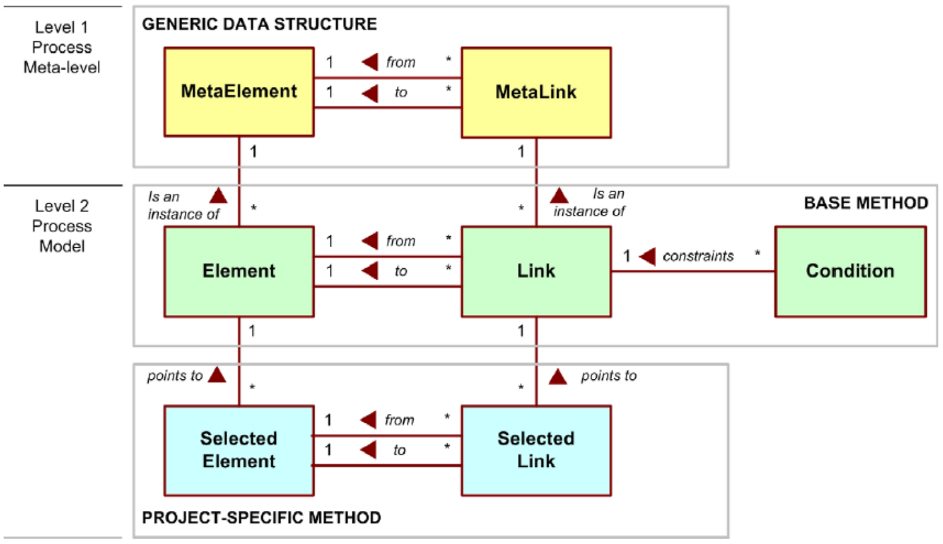


FIGURE 3.28: Generic data structure of the PCA [6]

and a project-specific method as a selection of the elements and links of the base method.

To perform the selection of elements from the base method, the PCA proposes the use of constraint rules. **Constraint rules** can be seen as assertions that restrict some aspect of the PCA for constructing project-specific methods. For instance, the rule "the deliverable Business Model depends on the activity Business modeling" indicates that, if the deliverable is selected, then the activity must be selected as well. This kind of rules allow the PCA to automate the configuration process. The algorithm that supports the PCA starts by selecting an element of the base method (typically, a starting activity) and ends when there is no link that connects the current element further with any other element.

The PCA is supported by a CAME environment that is called Agile Methodology Toolset (AMT) [37]. Figure 3.29 shows the general architecture of AMT. As the figure shows, AMT consists of six interconnected modules: MethAdapt, MethElicit, MethModel, MetEval, MethGen, and MethUse. The *MethAdapt* module facilitates method configuration applying constraint rules. *MethElicit* and *MethModel* are used for method modeling. While the former supports the definition

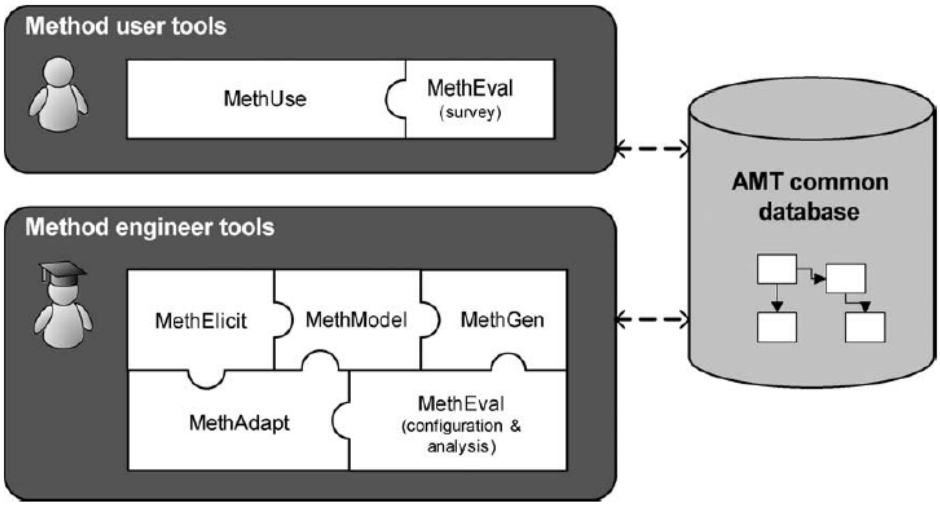


FIGURE 3.29: High level architecture of the AMT toolset [37]

of metamodels, the latter provides a graphical editor for metamodel instantiation. The main purpose of *MethEval* is to facilitate continuous evaluation of the method. The evaluation is carried out in a series of surveys that are conducted among all of the method users. *MethGen* represents a module for generating method reports. Finally, *MethUse* is utilized by method users who require access to the electronic method reference guide. The main purpose of this module is to make access to the method content as easy and quick as possible.

Analysis of the Proposal

One important advantage of the PCA is that it achieves a high level of automation thanks to the use of constraint rules. In contrast to this advantage and similarly to the MMC, the PCA has the following limitation:

- *Incomplete lifecycle coverage*: the PCA is exclusively focused on the conceptual configuration of methods so that they meet particular project needs; thus, it only supports the method design phase of the Method Engineering lifecycle.

Table 3.14 summarizes the relevant information of the PCA according to the template that is presented in Table 3.1.

Table 3.14: Classification of the Process Configuration Approach

Process Configuration Approach		
Specification language(s)	**Name**	x
	Type	x
	Formality level	x
	Size	x
Lifecycle coverage		MD
Perspective		x
Software support	**Name**	AMT
	Type	CAME
	Use	Research
	Automation level	High

3.3 Conclusions

This chapter presents a state-of-the-art review of Method Engineering. The review analyzes the most significant approaches that deal with the design, implementation, and execution of software development methods. Thus, the approaches of the review were not selected only due to their relevance in the Method Engineering field but also due to their closeness to the main topic of this thesis. In the review, the approaches are classified in three main categories according to their underlying strategy: assembly-based, paradigm-based, and configuration-based. For each approach, the review presents a summary of the approach together with an analysis of its main limitations.

In order to extract common limitations and disclose the potential contributions of this thesis, we analyzed all of the approaches in conjunction. As a result, we identified four main limitations. Below, we describe these limitations and outline the contributions of this thesis with respect to these issues.

High complexity. The state-of-the-art review that is presented in this chapter illustrates that Method Engineering approaches are, in general, complex to apply mainly due to the extensive knowledge that is required to put them into practice. This is especially true for assembly-based and paradigm-based approaches since they are purely manual. On the other hand, configuration-based approaches ease the burden of method engineers since they are (semi)automatic. However, these approaches require the manual definition of a base method that must reflect the actual performance of an entire organization in all its projects, which can result in a large model that is difficult to understand and expensive to maintain.

In addition to the level of automation, another important aspect that affects the applicability of Method Engineering is the formality level of the languages. For instance, languages such as MEL, NATURE, Map, and MRSL are highly formal and, therefore, they seem more appropriate for academic environments. This problem is more significant in the case of textual languages (e.g., MEL and MRSL) since they require that method engineers work at a low level of abstraction.

In order to alleviate the inherent complexity of Method Engineering, we propose making use of MDE techniques, in particular metamodeling, model transformations, and models at runtime. MDE techniques allow the method engineer to work at a high level of abstraction and they also increase automation, which is generally acknowledged as one of the most challenging and unsolved issues of Method Engineering [3, 161].

It is important to note that the application of metamodeling in Method Engineering is not new. In assembly-based approaches, method parts are instances of classes that are defined in a metamodel. Configuration-based approaches depend on a base method, which is instance of a specific metamodel. Paradigm-based approaches are generally based on metamodeling. Nonetheless, we leverage method models going one step further. Specifically, our approach uses method models as input of model transformations that (semi)automatically generate supporting software environments (i.e., method implementations). To the best of our knowledge, none of the reviewed approaches applies model transformations for this purpose. Saeki discusses in [162] the role of model transformations in Method Engineering; however, in this work, the author proposes two transformational approaches (for, respectively, performing method assembly and obtaining formal method descriptions), which are not related to the use that is proposed in this thesis. In addition to model transformations, in this thesis we also leverage method models at runtime to provide support for method execution. By applying model transformations and models at runtime, we increase the value of method models in terms of how much functionality they deliver. Our goal is

Table 3.15: Summary of lifecycle coverage

	Design	**Impl.**	**Exec.**
Brinkkemper *et al.*	+	+/-	+/-
Prakash	-	+	-
Ralyté *et al.* (assembly)	+	-	-
OPF	+	-	-
Method Editor	+	+/-	+/-
Rolland *et al.*	-	+	-
MetaEdit+	-	+	-
Ralyté *et al.* (paradigm)	+	-	-
Work Product Pool	+	-	+
Deneckère *et al.*	+	-	-
MMC	+	-	-
PCA	+	-	-

twofold; first, we aim to reduce the effort that requires designing and implementing methods, and, second, we aim to assist software engineers during the process of software development.

Incomplete lifecycle coverage. Most of the reviewed approaches only support one of the phases of the Method Engineering lifecycle. This reality is illustrated in Table 3.15, which summarizes the lifecycle coverage that the approaches provide. The "+" symbols indicate phases that are supported by the approaches; the "-" symbols indicate the opposite; and the "+/-" symbols indicate phases that are supported with limitations. As the table shows, only the Work Product Pool approach supports more than one phase. The approach of Brinkkemper *et al.* and Method Editor also support more than one phase, but they present important limitations with respect to process executability and CASE tool support. The rest of the approaches either focus on the method design or the method implementation. Approaches such as MMC, PCA, or Ralyté's provide rich ways to design software development methods, but they do not offer CASE tool generation capabilities. In contrast, approaches such as MetaEdit+, Prakash's, or Rolland's focus on the construction of customized CASE environments, but they neglect the design of the supported methods.

In this thesis, we study the linkage between the design, implementation, and execution of software development methods. Our main goal is to bridge

TABLE 3.16: Summary of perspectives

	Product	**Process**
Brinkkemper *et al.*	+	+/-
Prakash	+	-
Ralyté *et al.* (assembly)	+	+/-
OPF	+	+/-
Method Editor	+	+/-
Rolland *et al.*	+	+/-
MetaEdit+	+	-
Ralyté *et al.* (paradigm)	+	+/-
Work Product Pool	+	-
Deneckère *et al.*	+	+/-
MMC	+	-
PCA	x	x

the gap between these three phases in order to provide a methodological approach and software infrastructure that equally encompass all of them.

Limited process support. Even though methods are composed of two interrelated aspects – product and process – most of the reviewed approaches are product-oriented. This is illustrated in Table 3.16, which summarizes the results of the review with respect to the *perspective* property. The "+" symbols indicate perspectives that are supported by the approaches; the "-" symbols indicate the opposite; the "+/-" symbols indicate perspectives that are supported with limitations; and the "x" symbols indicate perspectives that do not apply to the approach under study. As the table shows, none of the approaches fully supports both perspectives, while all of them provide adequate coverage of the product perspective. Product-oriented approaches provide precise solutions to define the artifacts to be produced during the method execution and also how these artifacts must be created and manipulated. However, these approaches provide limited support for the specification and the execution of the process part of methods. In general, assembly-based approaches (such as those by Ralyté or Brinkkemper) define the development process when the method parts are assembled, generally by means of precedence relationships. Thus, the resulting process is limited in terms of control flow since complex behavior cannot be specified. Additionally, these approaches typically fail to address the producer aspects of methods, which are indirectly represented by means of attributes of the proposed method parts. On the other hand, in paradigm-based and configuration-based approaches, the process depends on the underlying metamodel. Nonetheless, most of the metamodels that are used in Method Engineering are product-oriented (e.g., GOPRR, MVM, and UML). In addition to these metamodels, various standard metamodels have recently appeared (e.g., ISO/IEC 24744 and SPEM 2.0). While these standards do support the specification of the process part of methods, they provide limited support for the definition of processes with a complex control flow. Furthermore, they do not formalize execution semantics, which hinders the automated execution of the processes that are defined using these languages.

In this thesis, we study how Process Modeling Languages can be used in the context of Method Engineering in order to improve method definitions in terms of process specification and process executability. In particular, we advocate the use of BPMN 2.0 in combination with SPEM 2.0, since SPEM 2.0 provides adequate support for method modeling, while it also offers powerful mechanisms for enhancing process definitions via behavioral modeling formalisms (such as BPMN 2.0). The combination of SPEM 2.0 and BPMN 2.0 is further justified in Appendix A, which provides a comparative analysis of these two languages.

Lack of software support. All of the Method Engineering approaches that are analyzed in this chapter have contributed to establish a solid and wide theoretical basis in the Method Engineering field. To put this theory into practice, some approaches have also been implemented as CAME or metaCASE environments. However, the use of this kind of tools has not been as widespread as expected partly because most of them still represent academic prototypes – up to now, MetaEdit+ is the only one that has been commercialized – that do not cover the entire lifecycle of Method Engineering. In general, CAME environments (such as Method Composer and MC Sandbox) focus exclusively on the method design, while metaCASE environments (such as MetaEdit+ and MERU) neglect this phase but provide advanced engineering techniques for method implementation. These differences in focus have created a situation where the Method Engineering field still lacks proper software support that equally enables the design, implementation, and execution of software development methods.

In order to develop a software environment that supports these three phases, in this thesis we analyze the requirements for such a tool and define a software architecture based on these requirements. The architecture has also been implemented in the context of Eclipse. The resulting CAME environment is called MOSKitt4ME.

Chapter 4

A Model-Driven Approach for Method Engineering

This chapter introduces a novel Method Engineering approach that aims to overcome the four limitations that are identified in Chapter 3 (see Section 3.3): high complexity, incomplete lifecycle coverage, limited process support, and lack of software support. To meet this challenge, our methodological approach advocates the use of Model-Driven Engineering (MDE) techniques. The combination of MDE and Method Engineering, which we refer to as model-driven Method Engineering, involves the intensive use of models to support three phases of the Method Engineering lifecycle: design, implementation, and execution. We believe that the use of MDE reduces the complexity of Method Engineering (which in turn positively affects its usefulness [163, 164, 165]) because it allows method engineers to work at a high level of abstraction and it also increases automation and reuse. In our approach, the level of abstraction is raised by allowing method engineers to design methods as models using a DSL that provides high-level conceptual constructs. On the other hand, automation is increased by means of model transformations, which take the method models as input and automatically generate method implementations; these method implementations are integrated environments that use the method models at runtime to assist software engineers during the execution of development projects. Finally, reuse is increased by allowing method/software engineers to encapsulate (in reusable assets) parts of the method models and also components of the supporting software environments. Thus, we enable the rapid construction of method models and we also automate the implementation of methods via the composition of reusable software components.

Our methodological approach supports the two parts that comprise methods – product and process – without neglecting the method participants and the method tools. To support the process part of methods, our approach prioritizes the specification of the method workflow, which determines the sequence of tasks to be performed by software engineers as well as the participants that are involved in each of these tasks. For the specification of the product part, our approach allows method engineers to associate products (which are defined in the method models by means of conceptual primitives such as "product" or "artifact") with *reusable assets* (which are stored in a repository). Depending on the asset type, these associations have different meaning. For instance, if a product is associated with an asset that contains a metamodel, then the asset specifies the product structure; on the other hand, the assets that encapsulate software tools (e.g., graphical editors or code generators) enable the specification of the tools that will support the creation of the products during the method execution.

In order to provide software support for the methodological approach that is presented in this chapter, we define a supporting software architecture and we also implement this architecture in the context of Eclipse. The resulting CAME environment is called MOSKitt4ME and is presented in detail in Chapter 5.

In a nutshell, the above discussion outlines the novel characteristics of our methodological approach. From these characteristics, we can generalize a key aspect that differentiates our approach from state-of-the-art Method Engineering: our approach envisions Method Engineering as a whole integrated discipline. This is in line with the idea of holism, which maintains that systems operate as wholes and their functioning cannot be fully understood solely in terms of their component parts [166]. By analogy to this idea, our approach aims to address all of the phases that comprise the Method Engineering lifecycle, supporting also all of the aspects that comprise software development methods. This is unlike current Method Engineering approaches, which in general try to explain Method Engineering in terms of its component parts, thereby neglecting the relationships that exist between these parts. For instance, the approaches that are based on metaCASE technology (see, e.g., [7, 40]) are exclusively focused on the technical aspects of Method

Engineering; thus, these approaches support the implementation of software development methods but they do not address the relationships that exist between these implementations and their conceptual counterparts. Other examples are approaches such as those by Karlsson *et al.* [4, 157] and Gonzalez-Perez *et al.* [22], which view methods through an exclusively product-oriented perspective, thereby neglecting the overall process of software development.

Due to the relevance of viewing Method Engineering in a holistic manner, this chapter is structured according to the phases that comprise the Method Engineering lifecycle: design, implementation, and execution. Prior to describing how our approach supports these phases, Section 4.1 provides an overview of the approach. Then, the phases of method design, method implementation, and method execution are detailed in Sections 4.2, 4.3, and 4.4, respectively. In each of these sections, we put special emphasis on how our approach supports the product and process aspects of software development methods. Finally, Section 4.5 draws some conclusions about the present chapter. For two examples of use of our methodological approach, we refer the reader to the industrial case study that is described in Appendix B and also to the technical report that can be found in [167].

4.1 Overview

In this section, before detailing our methodological approach, we provide a general overview. Specifically, this section outlines the origins of the approach (i.e., the context where our approach emerged) and the process that we followed for its development. Additionally, we provide an overview of the three phases that comprise the approach.

4.1.1 Origins: The MOSKitt Project

Our methodological approach emerged in the context of the MOSKitt project, which constitutes a collaborative work between the PROS research center and the CITMA to develop an Eclipse-based environment that supports the gvMétrica method (see Section 1.5). The result of the project was the MOSKitt tool, whose last public version was released at the end of 2012.

As Section 2.3.5 describes, MOSKitt has two important characteristics. First, it is built on Eclipse; therefore, it implements the Eclipse plug-in architecture. This architecture allows MOSKitt to be easily reconfigured and/or extended with new tools. Second, MOSKitt incorporates a wide range of tools for software systems analysis/design (such as graphical editors for defining UML 2.0 models and database schemas) and code generation (in different languages such as Java and PHP) as well as metatools that enable the construction of tools for software development (e.g., frameworks such as FEFEM, which enables the rapid creation of form-based editors). The infrastructure of MOSKitt also provides tools that facilitate the synchronization and navigation of models as well as the definition and execution of model transformations. Thus, MOSKitt embodies not only a CASE environment that supports a development method (i.e., gvMétrica), but it also offers an extensible platform for building CAME and CASE environments.

Despite the potential of MOSKitt, practitioners from the CITMA were still disappointed with (1) the high effort that was required to adapt the tool to changes in gvMétrica and (2) the lack of appropriate support for method execution. To solve these problems, we took advantage of the plug-in architecture of MOSKitt and developed MOSKitt4ME: a tool that is built upon MOSKitt and implements the model-driven Method Engineering approach that is detailed in this chapter. MOSKitt4ME alleviates the problems of MOSKitt because (1) it increases the degree of automation that is achieved in the construction and adaptation of methods and CASE environments, and (2) it provides assistance that allows software engineers to execute methods more easily.

4.1.2 Developing the Methodological Approach

The problems that were identified in MOSKitt imposed a mandatory requirement: our approach (and, therefore, the MOSKitt4ME environment) had to support not only the method design phase of the Method Engineering lifecycle, but also the implementation and execution phases. With this requirement in mind, we developed our approach following a two-step process.

In the first step of the process, we focused on the technical aspects of the Method Engineering lifecycle; that is, we determined how software environments had to be structured to support the methods defined by method engineers. To this end, we analyzed how MOSKitt is structured to support gvMétrica. In this analysis, we noted that we could leverage the plug-in architecture of MOSKitt by encapsulating the MOSKitt components in reusable assets; thus, these assets could be systematically used in the construction of new software environments (that support new versions of gvMétrica or other development methods).

Once we finished the analysis of MOSKitt, we performed the second step of the process, which deals with the conceptual aspects of the Method Engineering lifecycle. In this second step, we determined the method modeling language to be used by method engineers and found a way to bridge the gap between the concepts of this language and the technical data (i.e., the reusable assets). The result of this two-step process was a model-driven Method Engineering approach and a CAME environment that support three phases of the Method Engineering lifecycle: design, implementation, and execution. Section 4.1.3 provides an overview of each of these phases.

4.1.3 Phases of the Methodological Approach

Figure 4.1 graphically depicts the model-driven Method Engineering approach that is detailed in this chapter. The three phases that comprise the approach are summarized below.

Method Specification: the Method Design Phase

To support the method design phase, our approach enables the creation of a *method model* by means of a DSL that combines concepts from the SPEM 2.0 and BPMN 2.0 standards [18, 49]. The method model specifies (among other elements) the tasks to be carried out during the execution of the method, the workflow that establishes the execution order of these tasks, the participants that take part in this workflow, and the work products to be developed to reach the final software system. To enable the rapid creation of the method model, our approach allows method engineers to retrieve and assemble *conceptual* method parts (e.g., tasks, roles, and products) from a *repository* of reusable assets. This repository also stores *technical* assets, which encapsulate software tools such as MOSKitt components (e.g., the UML 2.0 editor or the Dashboard component) or dedicated tools created by means of the MOSKitt metatools (e.g., a form-based

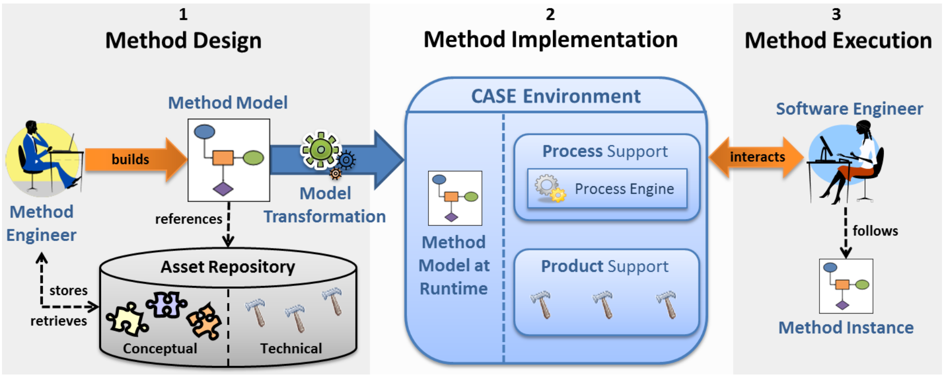


FIGURE 4.1: Overview of the methodological approach

editor created by means of FEFEM to support an organization-specific DSL). The tools that are stored in the repository allow method engineers to define the method technical data (i.e., the specific tools that will be involved in the method execution). To define this technical data, method engineers must link technical assets with method elements. A technical asset that is linked to a method element indicates the tool that will support the element during the method execution; for instance, a UML editor that is linked to a product called "Class model" will support the creation of specific instances of this product (i.e., specific UML class models).

Generating Software Support: the Method Implementation Phase

During the method implementation phase, a *CASE environment* that supports the method is automatically obtained by means of a *model transformation*. This CASE environment includes a *process engine* as well as software support for the creation and manipulation of the method *products*. The process engine (which is always included in the CASE environment regardless of the method that has been specified) provides method process support: it interprets the *method model at runtime* in order to assist software engineers during the entire process of software development. The software support for the method products is obtained from the technical assets that were linked to the method elements during the design phase. Thus, by means of the process engine and the tools obtained from the technical assets, the CASE environment incorporates support for both the product and process parts of the method.

Bringing Methods into Enactment: the Method Execution Phase

The method execution phase involves the enactment of *method instances* (in specific software development projects) using the CASE environment that was obtained in the previous phase. In this CASE environment, the method execution is driven by the process engine, which indicates the tasks that are executable according to the state of the projects as well as the tools that must be used in these tasks. When the tools do not require human participation (e.g., model transformations), the process engine automatically starts their execution; in contrast, when the tools require the participation of software engineers (e.g., graphical or textual editors), the CASE environment provides guidance on the use of the tools. This functionality allows software engineers to follow the method workflow more easily and it also partially automates the method execution phase. In addition to the process-related assistance, the CASE environment also allows software engineers to keep track of the method products; to this end, the CASE environment displays a hierarchical view that categorizes the products in domains, subdomains, and work product elements (according to the categories that are defined in the method model).

4.2 Method Design

In this section, we focus on the design phase of our methodological approach. The idea that lies behind this phase was conceived with a primary goal in mind: allowing methods (i.e., the method models that are produced in the design phase) to go beyond design so that they can be used to (1) automate the construction of the supporting software environments and also to (2) enable the execution of methods in these environments; thus, our approach can support the method design, implementation, and execution phases. To achieve this goal, method models must meet two requirements:

1. They must cover the product and process aspects of Method Engineering; thus, method models will contain enough information to be used as input of model transformations that automate the method implementation phase.
2. The process part of the method models must be executable; thus, the software environments that are obtained in the implementation phase will be

able to interpret the method model at runtime during the phase of method execution.

In order to meet the first requirement, our approach must provide method engineers with modeling primitives that allow them to specify the method product and process parts. With respect to the process part, method engineers need primitives for defining, inter alia, the method tasks, their execution order, and the performing roles. As for the product part, method engineers need to define the artifacts to be produced during the method execution and the tools that enable their creation (i.e., the method technical aspects).

In the literature, we can find several standards that offer this kind of primitives; most notably, SPEM 2.0 and BPMN 2.0. The former provides suitable product-related concepts (e.g., *Work Product Definition*, *Domain*, and *Tool Definition*), while the latter offers richer process-related primitives (e.g., gateways and different types of tasks). In view of this situation, we defined a DSL that integrates concepts from SPEM 2.0 and BPMN 2.0, and, thus, it reaps benefits from both languages (see Appendix A). This integration allowed us to obtain a DSL that includes the key method modeling constructs of SPEM 2.0 and also a set of BPMN 2.0 concepts that overcome the SPEM 2.0 limitations with respect to process specification.

Another important advantage of combining two languages into a DSL is that we enable the definition of constraints that apply to concepts from both languages; additionally, since methods are defined in single models, it is not necessary to keep various models synchronized. Even though this synchronization could be realized by means of bidirectional transformations, this type of transformations are still challenging and complex to implement nowadays [168, 169].

Below, we present our DSL in Section 4.2.1. To enable the rapid construction of methods using this DSL, our approach allows method engineers to utilize reusable method parts (e.g., roles, tasks, and products); our approach also enables the use of other types of reusable assets (e.g., software tools) to fully specify the product part of methods. Reusability is the focus of Section 4.2.2. Finally, Section 4.2.3 defines a three-step process that aims to assist method engineers during the entire phase of method design. This process has been conceived to foster reusability and also to allow method engineers to obtain executable methods (thereby meeting the second requirement that is described above).

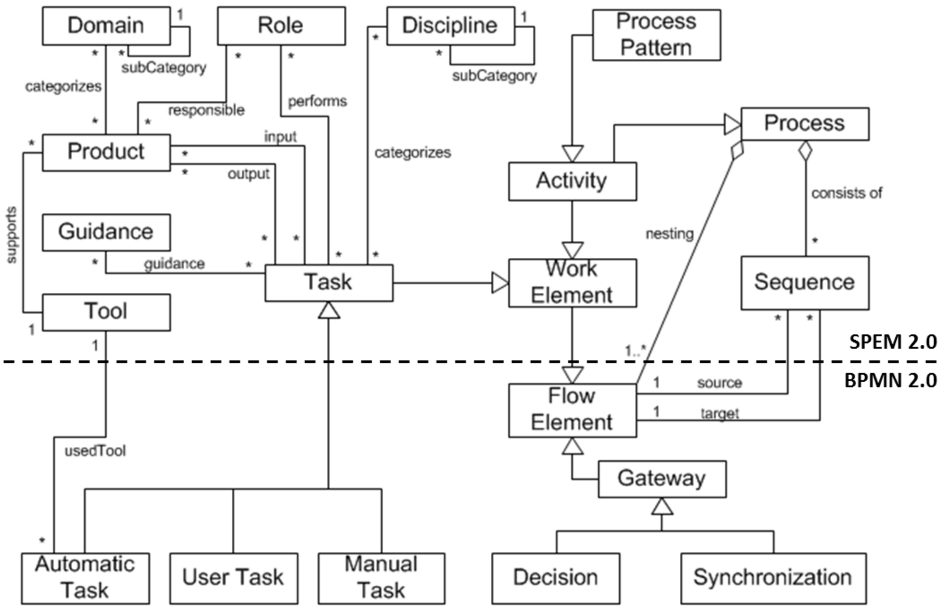


FIGURE 4.2: Abstract syntax of a DSL for method design

4.2.1 A DSL for the Conceptual Modeling of Methods

This section defines the abstract syntax of our DSL. As for the concrete syntax, we adopt the notation that is defined by SPEM 2.0 and BPMN 2.0; thus, we lower the barrier for domain experts who are already familiar with these standards. The abstract syntax of our DSL is defined by the metamodel that is shown in Figure 4.2. This metamodel is divided into two main sections. While the upper section contains concepts that are equivalent to SPEM 2.0 concepts, the lower section contains concepts that have their counterpart in BPMN 2.0. To illustrate the concepts of our DSL, we take examples from the case study that is presented in Appendix B.

Process is the central concept of the metamodel. The method processes are composed of *Flow Elements* and *Sequences*. Flow elements are the objects that appear in a process: *Tasks*, *Activities*, and *Gateways*; sequences connect flow elements to establish their execution order. Tasks and activities represent *Work Elements* (i.e., units of work to be carried out), while gateways establish mechanisms for enabling/disabling workflow paths. Gateways can be of two types: *Decisions* and

Synchronizations. Decisions create alternative paths within a workflow; synchronizations are used to create parallel paths of execution. Tasks can be of various types: automatic, user, and manual. *Automatic tasks* are executed without user participation; for instance, the task "Database model generation" involves the automatic construction of a database model by means of a model transformation. *User tasks* are carried out by the user by means of a software tool; for instance, the task "Database model revision" concerns the revision of the database model by means of a graphical editor. *Manual tasks* do not require any type of software; for instance, the task "Design validation" involves the manual validation of the work that is carried out by the analysts. In general, tasks can be grouped in categories (which are called *Disciplines*) and performed by *Roles* (e.g., "Analyst"). To assist the task performers, tasks can be associated to *Guidance* (e.g., textual descriptions or checklists). Tasks also typically consume and produce artifacts (which are represented by means of the input and output *Products* of the tasks); for instance, the task "Database model revision" has one input – "DB model" – which is also the output of the task. Products can be categorized by *Domain*, created by means of software *Tools* (e.g., "UML 2.0 class editor"), and assigned to roles (i.e., the people responsible for the products). Finally, tasks can be nested[1] within activities (e.g., "Business Logic Design"), which in turn represent complete processes. Activities can also be nested within other activities to create work breakdown structures. Thus, activities represent work units of coarse-grained granularity (when compared to the granularity level of the tasks). A special kind of activity are *Process Patterns*, which represent activities intended for systematic reuse.

All of the concepts that are summarized above have their counterpart in the SPEM 2.0 and BPMN 2.0 metamodels. The mappings between the concepts of our DSL and the concepts of SPEM 2.0 are summarized in Table 4.1. On the other hand, Table 4.2 displays the mappings between our DSL and BPMN 2.0.

4.2.1.1 Supporting the Method Product and Process Parts

When we conceived our DSL, we took into consideration that the DSL had to support the process part of methods without neglecting the product part. Below,

[1]Nesting is allowed (by inheritance) by the *nesting* relationship, which establishes that processes can contain flow elements.

Table 4.1: Mappings between concepts of our DSL and SPEM 2.0

DSL	**SPEM 2.0**
Domain	Category (kind=Domain)
Discipline	Category (kind=Discipline)
Product	Work Product Definition
Guidance	Guidance
Tool	Tool Definition
Task	Task Definition
Role	Role Definition
Work Element	Work Breakdown Element
Activity	Activity
Process Pattern	Activity (kind=Process Pattern)
Process	Activity (kind=Delivery Process)
Sequence	Work Sequence

Table 4.2: Mappings between concepts of our DSL and BPMN 2.0

DSL	**BPMN 2.0**
Automatic Task	Service Task
User Task	User Task
Manual Task	Manual Task
Flow Element	Flow Element
Gateway	Gateway
Decision	Exclusive Gateway
Synchronization	Parallel Gateway

we identify the concepts of SPEM 2.0 that we selected to support both method parts; then, we illustrate how BPMN 2.0 allowed us to overcome the limitations of SPEM 2.0.

Product. The product part of methods is represented in our DSL by the *Product* concept, together with *Domain* (which categorizes products) and *Tool* (which establishes how the products must be created during execution). An example of product is "Process Model", which can be categorized in a domain called "Models" and created by means of a tool called "BPMN Editor".

Process. Our DSL provides concepts for defining the overall process of methods (i.e., *Task*, *Activity*, *Process Pattern*, *Discipline*, and *Role*); it also uses the

Guidance concept to denote the micro-processes that specify how to produce single method products. An example of task is “Create Class Model”, which can be contained in an activity called “Information System Design”, categorized in a discipline called “Modeling”, and performed by a role called “Designer”. This task can also be associated, e.g., to a process model that provides guidance on the creation of UML 2.0 class diagrams.

Even though SPEM 2.0 offers primitives that enable the specification of methods in a clear, intuitive, and natural way, it also presents an important limitation: SPEM 2.0 provides limited support to represent complex processes since it only allows method engineers to establish precedence relationships between tasks. To overcome this limitation, we selected a set of BPMN 2.0 concepts that endow our DSL with richer process-related primitives. Specifically, we added to our DSL the concepts of *Gateway*, *Decision*, and *Synchronization*; thus, we enable the specification of more complex workflows (i.e., those that can incorporate branching conditions as well as synchronizations of different execution paths). In addition to these concepts, we also added *Automatic Task*, *User Task*, and *Manual Task*; thus, our DSL enables the specification of different task types. These types depend on the tools involved in the process of software development (e.g., a task that does not require the use of any software tool will be considered as a manual task).

4.2.2 Promoting Reuse: A Method Fragment Taxonomy

It is a major goal of our methodological approach to foster reusability. For this reason, we define different types of reusable assets that enable rapid method design by means of their assembly. By analogy to the work of Brinkkemper *et al.* [5, 8] (see Section 3.2.1.1), we use the term **method fragment** to denote the reusable assets that are used in our approach. Even though Brinkkemper *et al.* define the term method fragment as “a description of a software development method, or any coherent part thereof”, our notion of method fragment slightly differs from this vision. Rather than distinguishing levels of granularity, our method fragments follow the definition that was given by Hofstede *et al.* in [28]: method fragments are “any coherent part of a metamodel”. This definition suggests that method fragments are created by instantiation from classes of a metamodel, and, thus, method fragments can contain any type of atomic method element. Nonetheless, there is a general agreement that the most important kinds of method fragments are: products, processes, and roles (or producers) [26]. For this reason, our approach supports, among others, these three types of method fragments.

The notion of method fragment contrasts with other types of method building blocks that (rather than containing atomic elements) represent composites that encapsulate two or more related method parts [26]. One example of these composites are the method chunks that are proposed by Ralyté *et al.* in [129, 130] (see Section 3.2.1.3). Unlike method fragments, method chunks encapsulate a tight connection between a product-oriented fragment and a process-oriented fragment. This tight connection, however, presents an important disadvantage: such a process-product linkage is neither one-to-one nor unique in real-life scenarios [131]. This problem is solved by method fragments since they allow method engineers to link one process fragment with several product fragments (or vice versa). This increment in flexibility also prevents method engineers from replicating information in the repository, but it comes at the expense of increasing the effort that is required for method assembly (since a higher number of fragments will be required to obtain the final method). Nonetheless, this additional effort is alleviated in our approach by providing appropriate software tools. In summary, our methodological approach works with method fragments that support the highest possible precision in creating new methods while the slight increase in effort is resolved by means of appropriate tool support.

4.2.2.1 Fragment Types

We define a taxonomy that includes the types of method fragments that are supported in our approach. To determine this taxonomy, we took into consideration the thesis goals. Since a major goal of this thesis is to support both the conceptual and technical aspects of Method Engineering, our taxonomy defines two main types of method fragments: *Conceptual Fragments* and *Technical Fragments*. These two types of fragments correspond, respectively, to the conceptual and technical assets of Figure 4.1.

Figure 4.3 graphically depicts our method fragment taxonomy. In this taxonomy, the fragments that are colored in gray represent the actual fragments that are used for method assembly, while the rest of fragments, which are colored in white, are

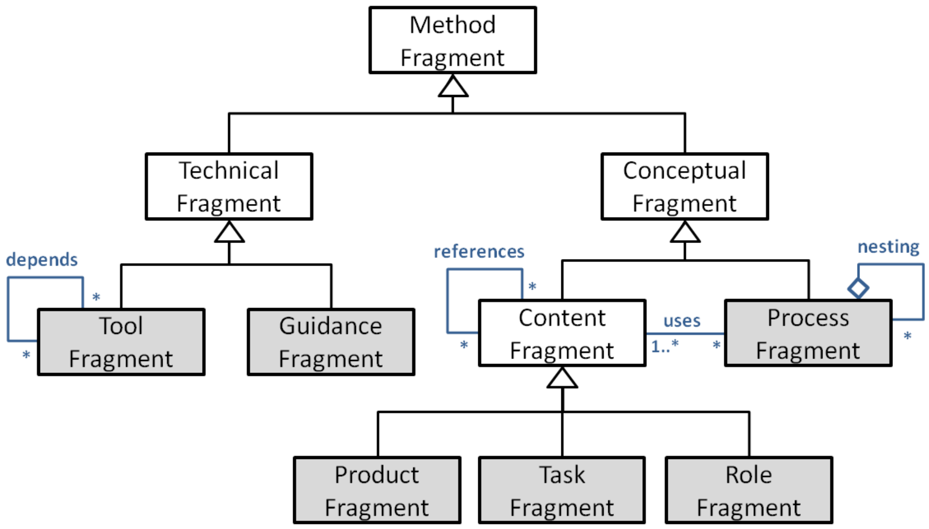


FIGURE 4.3: Method fragment taxonomy

only used for categorization purposes. Below, we describe conceptual fragments and technical fragments; then, we detail the relationships that can be established between the different fragment types.

Conceptual Fragments

Conceptual fragments encapsulate definitions of individual method parts (e.g., tasks, roles, and products); therefore, this type of method fragments are created by instantiation from concepts of our DSL (see Figure 4.2). Conceptual fragments can be of two types: *Process Fragments* or *Content Fragments*. This separation has been established according to the separation of method content and process that is proposed by the SPEM 2.0 standard (see Section 2.1.2.1). Specifically, process fragments contain definitions of reusable processes; these reusable processes are represented by the "Process Pattern" concept of our DSL. On the other hand, content fragments contain definitions of reusable method content. Since the main content elements of our DSL (in accordance with SPEM 2.0) are products, tasks, and roles, our methodological approach proposes three types of content fragments: *Product Fragments*, *Task Fragments*, and *Role Fragments*. These three types of content fragments store, respectively, instances of the "Product", "Task", and 'Role‘" concepts of our DSL.

Technical Fragments

In contrast to conceptual fragments (which encapsulate method elements that are created by instantiation from concepts of our DSL), technical fragments encapsulate CASE functionality. The functionality that is offered by CASE environments is generally implemented as software tools (such as textual/graphical editors, M2M transformations, and code generators) that enable the creation of method products (see Section 2.1.3). In addition to these tools, the Method Engineering community also gives special attention to the guidelines (e.g., textual descriptions, checklists, or process models) that assist software engineers during the product creation (see, for example, [134, 145]). These guidelines correspond to the small-scale processes that we describe in Chapter 1. To support the encapsulation of both software tools and guidelines, we included in our taxonomy two types of technical fragments: *Tool Fragments* and *Guidance Fragments*.

One important requirement for technical fragments is that they must implement a common interface so that the process engine (see Figure 4.1) can invoke them during the method execution. To illustrate this idea, let us consider that all of the technical fragments that encapsulate model transformations implement a common operation. Thus, during the method execution, the process engine can launch the model transformations by simply invoking this operation.

Note that, despite the existence of the "Guidance" concept in our DSL, guidance fragments do not contain an instance of this class. This is because we advocate a more tool-oriented vision of guidance. An example of tool-oriented guidance is the Eclipse dynamic context help. This type of guidance, which must be implemented as Eclipse plug-ins, is shown in the Eclipse "Help" view. This view allows the user to dynamically interact with the contextual help, which provides a more enriching experience than the one that is provided by, for example, static text. To complement this tool-oriented guidance, method engineers can also associate *Guidance* elements to method tasks. These elements, which instantiate the "Guidance" concept of our DSL, provide different types of guidelines, such as checklists, examples, reports, templates, and white papers [18].

Relationships Between Method Fragments

In addition to the various types of fragments that are supported in our approach, the taxonomy that is shown in Figure 4.3 also illustrates several relationships between method fragments. The *Uses* relationship represents that a process that is stored in a process fragment can use one or more elements contained in content fragments. This is in accordance with the approach that is adopted by SPEM 2.0, where processes take content elements and relate them into partially-ordered sequences that are customized to specific types of projects [18]. On the other hand, the *Nesting* relationship represents that a process may contain nested processes and the *References* relationship represents that a method content element may reference other method content elements (e.g., a task referencing its output work products). Finally, the *Depends* relationship establishes dependencies between the software tools that are encapsulated in tool fragments. A dependency relationship can be defined as an unidirectional association between two software tools, *T1* and *T2*, which implies that *T1* requires *T2* for its correct operation. Dependency relationships form dependency trees, where nodes represent software tools and all of the descendants of a node represent its dependencies. The dependencies that are defined between tool fragments are used during the method implementation phase; thus, all of the tools that are required to support the methods can be successfully integrated in the generated software environments.

4.2.2.2 Fragment Structure

With the aim of minimizing the costs of reuse of our method fragments, we decided to advocate consistent and standard packaging. For this reason, in our approach, method fragments are stored in a repository as reusable assets that conform to the Reusable Asset Specification (RAS) standard [170]. According to the guidelines that are provided by RAS, our method fragments are stored as *.ras* zip files, which contain a manifest file together with a set of artifacts that represent the fragment content. These artifacts can be, for instance, Eclipse plug-ins (in the case of technical fragments) or a model containing a single element (in the case of conceptual fragments). The manifest file is an XML document that contains a set of properties. These properties characterize method fragments, and, therefore, they allow method engineers to search and retrieve fragments from the repository.

Depending on the type of method fragment, the properties that are stored in the manifest file differ; nonetheless, there are some properties that are shared by all of the types of method fragments. These common properties are the following:

- *Name*: this property represents the identifier of the method fragment.
- *Type*: this property classifies the fragment in one specific category. For conceptual fragments, there are four possible types, as defined in our fragment taxonomy (see Figure 4.3): "Task", "Role", "Product", and "Process". For technical fragments, the *type* property represents the kind of tool/guidance that is encapsulated in the fragment. Specifically, there are eight possible types of tools: "Graphical Editor", "Metamodel", "Textual Editor", "Grammar", "Form-based Editor", "Model Transformation", "External Tool", and "Internal Tool". On the other hand, we only consider one generic type of guidance, which is simply represented by the "Guidance" type.
- *Origin*: this property establishes where the fragment content originates from. For instance, a conceptual fragment can contain a task that was extracted from the gvMétrica method, or a tool fragment can contain a graphical editor that was extracted from the MOSKitt platform.
- *Objective*: this property defines the purpose of the elements that are contained in the method fragment.
- *Description*: this property contains general information about the method fragment.

In addition to the above properties, tool fragments are also characterized by an interface that is composed of two main properties:

- *Input*: this property establishes the requirements that are needed to execute the software tool that is contained in the tool fragment. For instance, the input of a model transformation is its input model.
- *Output*: this property defines the artifacts that can be produced by means of the software tool that is contained in the tool fragment. For instance, the output of a model transformation is its output model.

4.2.3 A Process for Method Design

In this section, we propose a process to carry out the method design phase. This process was conceived with a twofold objective in mind: fostering reusability and

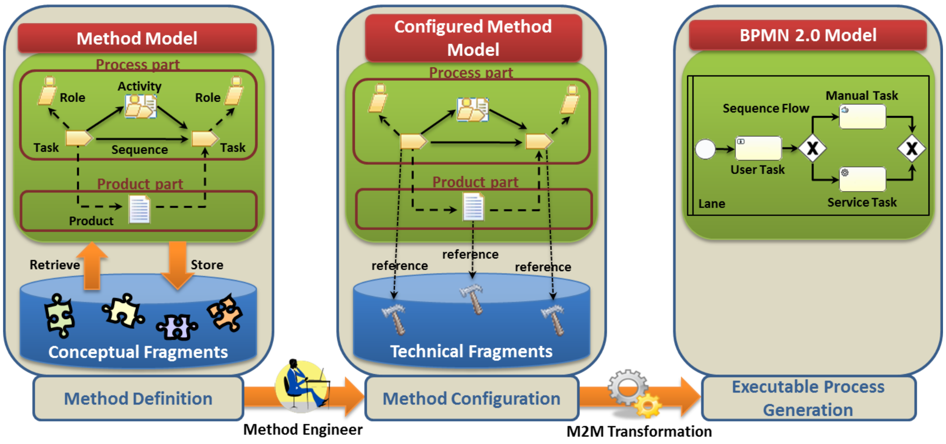


FIGURE 4.4: The method design process

enabling the construction of executable representations of the method models. Based on these goals, we propose a process that comprises the steps that are depicted in Figure 4.4: *method definition*, *method configuration*, and *executable process generation*. The first step of the process involves the construction of the conceptual definition of the method; that is, the method model, which instantiates the metamodel of our DSL (see Section 4.2.1). Then, in the second step of the process, method engineers employ reusable assets to specify technical details (i.e., the tools that will support the method during its execution in development projects). Separating these two steps allows method engineers to keep a generic definition of the method (which does not contain details about the specific languages, notations, and tools that will be used during the method execution) and to perform different configurations that meet the needs of different development projects. Finally, in the last step of the process, an executable representation of the method process part is automatically obtained by means of a M2M transformation. The executable process model that is obtained in this step is compliant with the BPMN 2.0 standard. The three steps of our process for method design are detailed in Sections 4.2.3.1, 4.2.3.2, and 4.2.3.3, respectively.

Table 4.3: Relationships between method elements and conceptual fragments

Method Element	**Relationship**	**Conceptual Fragment**
Task	≪input≫ / ≪output≫	Product
Task	≪performs≫	Role
Product	≪responsible≫	Role
Activity	≪nesting≫	Task
Activity	≪nesting≫	Process

4.2.3.1 Method Definition

In this step, method engineers build the conceptual definition of the method by means of the concepts that are provided by our DSL. The main concepts that comprise the process part of the method model are the *tasks*, *activities*, *sequences*, and *roles*. On the other hand, the main elements that comprise the product part are the method *products*.

During the construction of the method model, method engineers can reuse conceptual fragments by retrieving them from a repository. The elements that are contained in conceptual fragments can be integrated in the method under construction by means of the relationships that are provided by our DSL. Table 4.3 shows the relationships of our DSL that can be used to associate the elements of a method with the elements that are contained in conceptual fragments. The rationale of these relationships is the following:

Task – Product. The tasks that are defined in the method under construction can be associated to products contained in product fragments. Tasks and products can be associated by means of the ≪*input*≫ and ≪*output*≫ relationships that are provided by our DSL. The products that are set as input of a task define artifacts that are required so that the task can be properly executed. The products that are set as output define artifacts that are obtained after the task execution.

Task – Role. Similarly to products, the roles that are contained in role fragments can be associated to the tasks of a method. Tasks and roles can be associated by means of the ≪*performs*≫ relationship that is provided by our DSL. The roles that are set as performers of a task define sets of related skills and competencies that individuals must possess in order to properly carry out the task.

Product – Role. The products that are defined in the method under construction can be associated to roles defined in role fragments. Products and roles can be associated by means of the ≪*responsible*≫ relationship that is provided by our DSL. The roles that are set as responsible of a product define those individuals that are in charge of the product construction.

Activity – Task. The activities of a method can be associated to tasks defined in task fragments. Activities and tasks can be associated by means of the ≪*nesting*≫ relationship that is provided by our DSL. The tasks that are nested in an activity define units of work whose granularity level is lower than the granularity level of the activity.

Activity – Process. The activities of a method can be associated to process patterns defined in process fragments. Activities and process patterns can be associated by means of the ≪*nesting*≫ relationship that is provided by our DSL. The process patterns that are nested in an activity define reusable processes whose granularity level is lower than the granularity level of the activity.

4.2.3.2 Method Configuration

During the method configuration[2], the method engineer defines the technical details of the method that was defined in the previous step. These technical details represent the tools that will allow software engineers to perform the method tasks during the method execution as well as the guidance that will assist them during the tasks performance. In order to define this technical data, the method engineer must associate the tasks and products of the method with technical fragments that are stored in a repository.

Table 4.4 gathers all the associations that are allowed between method elements and technical fragments. The rationale of these associations is the following:

[2]Note that this step differs from the Method Engineering approach of method configuration, which was proposed by Karlsson *et al.* in [4].

Table 4.4: Relationships between method elements and technical fragments

Method Element	**Relationship**	**Technical Fragment**
Product	≪supports≫	Metamodel
Product	≪supports≫	Graphical Editor
Product	≪supports≫	Form-based Editor
Product	≪supports≫	Grammar
Product	≪supports≫	Textual Editor
Product	≪supports≫	External Tool
Product	≪supports≫	Internal Tool
Task	≪usedTool≫	Model Transformation
Task	≪guidance≫	Guidance

Product – Metamodel. The products of the method model can be associated to metamodels by means of the ≪*supports*≫ relationship that is provided by our DSL. A metamodel that is associated to a product defines the abstract syntax of the notation that allows software engineers to create the product during execution. Note that, when a method element is associated with a technical fragment, an instance of the “Tool” concept is also created. This instance is the element that references the technical fragment that is stored in the repository.

Product – Graphical Editor. The products of the method model can be associated to graphical editors by means of the ≪*supports*≫ relationship. A graphical editor that is associated to a product defines the abstract and concrete syntax of the notation that allows software engineers to create the product during execution.

Product – Form-based Editor. The products of the method model can be associated to form-based editors by means of the ≪*supports*≫ relationship. Similarly to graphical editors, a form-based editor that is associated to a product defines the abstract and concrete syntax of the notation that allows software engineers to create the product during execution. The difference with graphical editors lies in the nature of the graphical user interface. Unlike graphical editors, the interface of form-based editors is composed of graphical widgets such as tables, text fields, labels, radio buttons, and check boxes.

Product – Grammar. The products of the method model can be associated to grammars by means of the ≪*supports*≫ relationship. A grammar that is associated to a product defines the syntax of the textual language that allows software engineers to create the product during execution. Additionally, the technical fragments that contain grammars define the semantics of the language, which can be specified informally using natural language or formally, e.g, by means of an ontology.

Product – Textual Editor. The products of the method model can be associated to textual editors by means of the ≪*supports*≫ relationship. A textual editor that is associated to a product defines the syntax and semantics of the textual language that allows software engineers to create the product during execution. Additionally, the editor also defines its visual features (e.g., syntax coloring, automatic code formatting, and content assist).

Product – External Tool. The products of the method model can be associated to external tools by means of the ≪*supports*≫ relationship. An external tool is a software application that is installed in the system and cannot be integrated in the software environment supporting the method (e.g., because it has been implemented using a different technology). In this case, the software environment stores a reference to the external tool so that it can be opened during execution for the creation of the product. Unlike the other types of technical fragments, the technical fragments that define external tools do not encapsulate these tools but rather contain references to them.

Product – Internal Tool. The products of the method model can be associated to internal tools by means of the ≪*supports*≫ relationship. An internal tool is a software application that is already installed in the CAME environment (e.g., Eclipse frameworks such as GMF or EMF). The technical fragments that represent internal tools do not need to encapsulate the implementation of these tools; instead, the fragments must store the identifiers of the tool components (e.g., plug-ins in the context of Eclipse). When a CASE environment is generated, the internal tools that are associated to method products are included in the environment; thus, it will be possible to use the internal tools for the creation of the products. One important advantage of internal tools is that they are not subject to any requirement. For instance, unlike the other types of fragments, internal tools do not have to implement a common interface (see Section 4.2.2.1); therefore, they can represent any type of software tool. The disadvantage is that the process engine (see Figure 4.1) will not be able to launch them automatically when the method is being executed.

Task – Model Transformation. The tasks of the method model can be associated to model transformations by means of the ≪*usedTool*≫ relationship. A task that is associated to a model transformation is considered automatic and the execution of the task involves launching the model transformation.

Task – Guidance. The tasks of the method model can be associated to guidance (e.g., textual descriptions or process models) by means of the ≪*guidance*≫ relationship of our DSL. This guidance will be used to assist software engineers during the performance of the method tasks.

4.2.3.3 Executable Process Generation

In this step, method engineers obtain an executable representation of the method process part. To facilitate this task, we implemented a M2M transformation that takes the configured method model as input and automatically generates a process model. This process model conforms to the BPMN 2.0 standard; thus, we ensure that the model can be executed by a process engine. When the *Executable Process Generation* step is finished, method engineers therefore keep two separate models. One model defines the method in terms of concepts from our DSL, while the other model stores (in terms of BPMN 2.0) an executable representation of the method process part. These two models are connected by means of trace links, which associate the BPMN 2.0 elements (that were automatically generated) to the method elements from which they originate. The reason for keeping two models is that they complement each other. On the one hand, the BPMN 2.0 model stores an executable representation of the method – this model needs not be edited manually. On the other hand, the method model contains information about the method that cannot be represented in BPMN 2.0 (e.g., the method roles and tools).

In order to provide further insight on how the M2M transformation obtains the BPMN 2.0 model from the configured method model, we summarize in Table 4.5

TABLE 4.5: Mappings between SPEM 2.0 concepts of our DSL and BPMN 2.0

	DSL (SPEM 2.0)	**BPMN 2.0**
1	Process	Process
2	Activity	Process and Call Activity
3	Process Pattern	Process and Call Activity
4	Sequence	Sequence Flow
5	Role	Lane

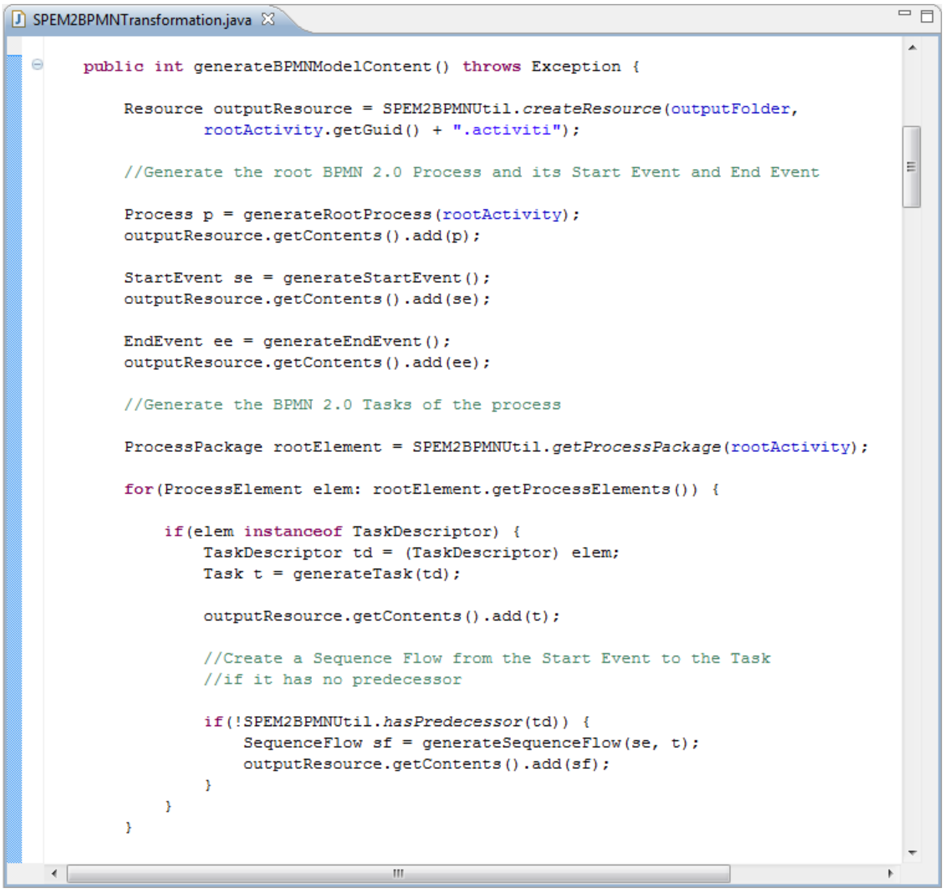
some of the mappings between the concepts of our DSL and BPMN 2.0. Note that this table only contains mappings for the concepts of our DSL that belong to the process part of methods. Additionally, Table 4.5 only contains mappings for the concepts of our DSL that are equivalent to concepts of SPEM 2.0; the mappings of the rest of the concepts can be examined in Table 4.2. The description of the mappings of Table 4.5 is the following:

1. An element of type *Process*, which represents the root element of the method model, is mapped into a BPMN 2.0 *Process*.

2. An element of type *Activity* is mapped into two different elements: a BPMN 2.0 *Call Activity* and a BPMN 2.0 *Process*. When the *Call Activity* is executed, it invokes the BPMN 2.0 *Process*, which is the root element of a separate process. Thus, we emulate in BPMN 2.0 the work breakdown structures that are formed in the method model when activities are nested within other activities.

3. Similarly to the *Activities*, an element of type *Process Pattern* is mapped into two different elements: a BPMN 2.0 *Call Activity* and a BPMN 2.0 *Process*. The difference with activities lies in the fact that the BPMN 2.0 *Process* is only generated once, regardless of the number of times that the *Process Pattern* is applied in the method model. This is a way of promoting reusability since the BPMN 2.0 model may contain many different *Call Activities* that invoke the same BPMN 2.0 *Process*.

4. An element of type *Sequence* is mapped into a BPMN 2.0 *Sequence Flow*. The source of the *Sequence Flow* is set to the BPMN 2.0 element that is generated from the *predecessor* of the *Sequence*; the target is set to the BPMN 2.0 element that is generated from the *successor* of the *Sequence*. Note that,



```
public int generateBPMNModelContent() throws Exception {

    Resource outputResource = SPEM2BPMNUtil.createResource(outputFolder,
            rootActivity.getGuid() + ".activiti");

    //Generate the root BPMN 2.0 Process and its Start Event and End Event

    Process p = generateRootProcess(rootActivity);
    outputResource.getContents().add(p);

    StartEvent se = generateStartEvent();
    outputResource.getContents().add(se);

    EndEvent ee = generateEndEvent();
    outputResource.getContents().add(ee);

    //Generate the BPMN 2.0 Tasks of the process

    ProcessPackage rootElement = SPEM2BPMNUtil.getProcessPackage(rootActivity);

    for(ProcessElement elem: rootElement.getProcessElements()) {

        if(elem instanceof TaskDescriptor) {
            TaskDescriptor td = (TaskDescriptor) elem;
            Task t = generateTask(td);

            outputResource.getContents().add(t);

            //Create a Sequence Flow from the Start Event to the Task
            //if it has no predecessor

            if(!SPEM2BPMNUtil.hasPredecessor(td)) {
                SequenceFlow sf = generateSequenceFlow(se, t);
                outputResource.getContents().add(sf);
            }
        }
    }
```

FIGURE 4.5: Excerpt of the M2M transformation

for all of the *Tasks* of the method model that are not *successor* of any *Sequence*, a BPMN 2.0 *Sequence Flow* will be created connecting the *Start Event* and the corresponding *Service/User/Manual Task*. Likewise, for all of the *Tasks* of the method model that are not *predecessors* of any *Sequence*, a BPMN 2.0 *Sequence Flow* will be created connecting the corresponding *Service/User/Manual Task* to the *End Event*.

5. An element of type *Role* is mapped into a BPMN 2.0 *Lane* if and only if the *Lane* has not been previously generated.

For illustrative purposes, Figure 4.5 shows the first instructions of the Java code that implements the M2M transformation in MOSKitt4ME. As the figure shows, the first action of the transformation is to create the output BPMN 2.0 model; that is, the *.activiti* file that is supported by the Activiti Designer editor. Then, the transformation generates the root process of the BPMN 2.0 model (see mapping 1 of Table 4.5). Once this process has been obtained, the transformation generates the start event and the end event. When these two elements are included in the output model, the transformation iterates the tasks of the method model. For each of these tasks, the transformation produces a BPMN 2.0 task, which will be a service task, a user task, or a manual task (see mappings of Table 4.2). Finally, the transformation generates a BPMN 2.0 sequence flow for each of the tasks that have no predecessor. The resulting sequence flows connect the tasks with the start event of the BPMN 2.0 model (see mapping 4 of Table 4.5).

4.3 Method Implementation

The method implementation phase involves the construction of an integrated software environment that supports the method that is defined during the method design phase. As Figure 4.6 illustrates, we have automated the method implementation by means of a *model transformation* that obtains a *CASE environment* from the *method model*. This model transformation makes use of the product and process parts of the method to obtain the final tool. The *product part* corresponds to the artifacts that must be created during the method execution and also to the tools that enable the creation and manipulation of these artifacts (i.e., the technical fragments); the *process part* corresponds to the BPMN 2.0 model that is obtained in the last step of the method design phase. Using the method product and process parts, the model transformation obtains a software environment that supports both aspects of the method.

In the software environment, the process part of the method is supported by a process engine for BPMN 2.0. The method process part will be deployed into this engine during the process of CASE environment construction. On the other hand, the technical fragments that are referenced in the method model are the components that provide product support in the software environment: these

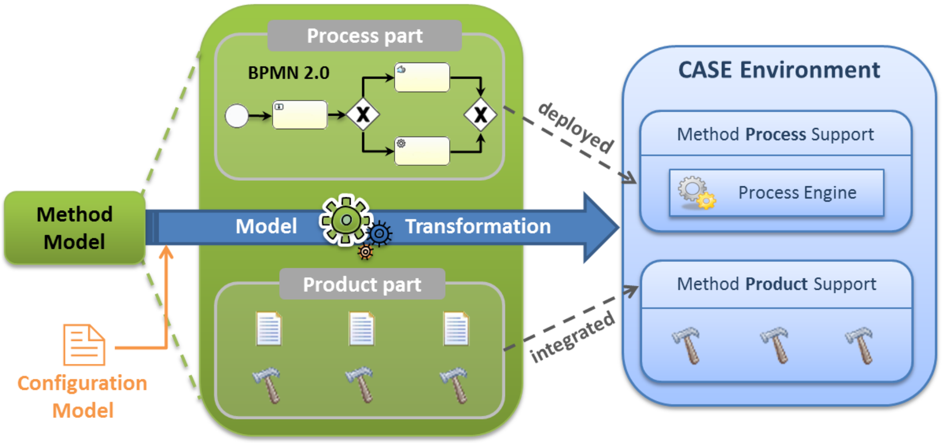


FIGURE 4.6: The method implementation phase

fragments are integrated in the environment to enable the creation of the method products.

The Configuration Model

With the aim of providing a higher level of flexibility in the construction of CASE environments, we introduce a new model in our proposal for method implementation: the *configuration model* (see Figure 4.6). This model stores information about the CASE environment (such as its identifier and name), and, therefore, it represents a mechanism for fine-tuning the model transformation (and, in turn, the CASE environment) according to specific contextual needs. The benefits of using configuration models were illustrated by Wagelaar *et al.* in [171]. This work presents a comparative study of different techniques (such as feature models and knowledge-based systems) for managing the configuration of model transformations.

To obtain the configuration model, method engineers can either generate (automatically) a default version of the model or create it manually. In both cases, method engineers can edit this model (if necessary) to specify attributes of the CASE environment, such as its identifier, name, target path, about information, or version. This is illustrated in Figure 4.7, which shows the general structure of the configuration model. In addition to these attributes, method engineers can also specify further information by means of general *properties*, such as the

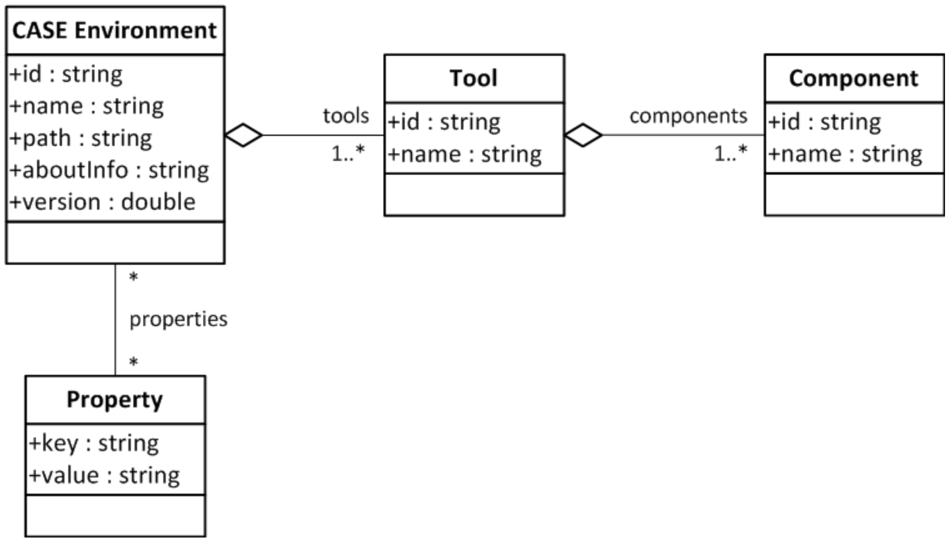


FIGURE 4.7: General structure of the configuration model

splash screen, the window title, the window images, the welcome page, or the initial window dimensions. All of this information must be specified by method engineers prior to the execution of the model transformation. During the process of CASE environment construction, the method model will be used to complete the configuration model with the *tools* and *components* that will comprise the CASE environment. This data (which is obtained from the technical fragments that are referenced in the method model) allows the configuration model to become the place where all the information about the CASE environment is uniquely gathered. Thus, we facilitate the last step of the construction process, where the configuration model is used to generate the final tool. This process is detailed in Section 4.3.1.

4.3.1 An Automatic Process for CASE Environment Construction

This section details the process that is implemented in our approach to support the automatic construction of CASE environments. This process is graphically depicted in Figure 4.8. As the figure shows, the process comprises three sequential steps: *identification of software tools*, *resolution of dependencies*, and *deployment*

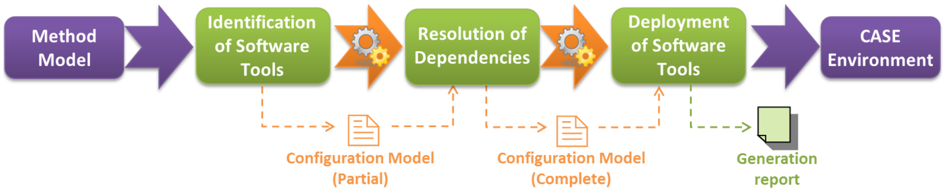

Method Model
Identification of Software Tools
Resolution of Dependencies
Deployment of Software Tools
CASE Environment
Configuration Model (Partial)
Configuration Model (Complete)
Generation report

FIGURE 4.8: CASE environment construction process

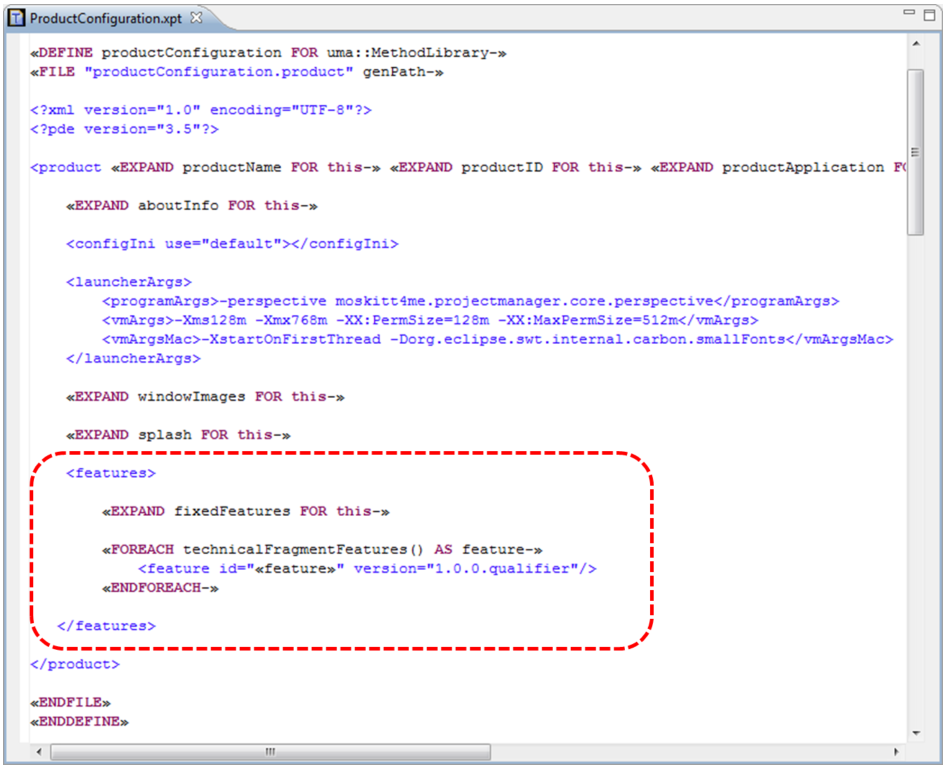
of software tools. While the first two steps aim to obtain a complete version of the configuration model, the last step is in charge of the generation of the final software environment.

More specifically, the first step of the process involves the identification of the software tools that must be integrated in the final software environment to support the product and process parts of the method. Once these tools are identified, their software dependencies are resolved in the second step. All of the tools that are identified in the first two steps are used to complete the configuration model. Finally, in the third step, the configuration model is used to generate the software environment that supports the execution of the method. These three steps are detailed in Sections 4.3.1.1, 4.3.1.2, and 4.3.1.3, respectively.

4.3.1.1 Identification of Software Tools

This step involves the identification of the tools that must be integrated in the final software environment so that it provides support for the method. To identify these tools, the model transformation explores the method model. For each task and product of the method, the transformation checks whether the task or the product is associated to a technical fragment. If so, the model transformation includes – in the *configuration model* – the identifier and name of the software tool that is contained in the fragment. Once the transformation has completely explored the method model, an additional tool is added to the configuration model. This tool is the process engine, which is always deployed in the CASE environments regardless of the method that has been specified.

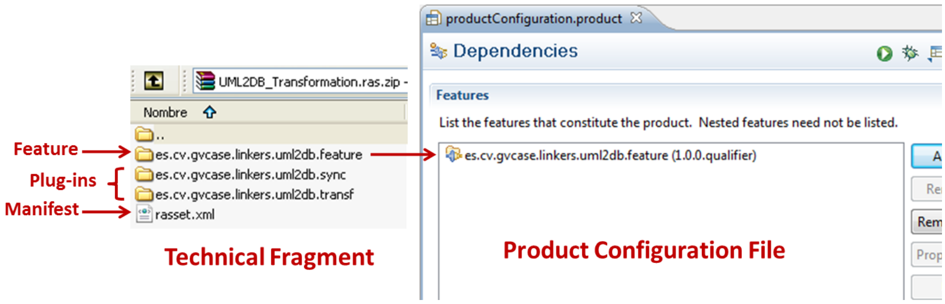
For illustrative purposes, Figure 4.9 shows an excerpt of a Xpand template that implements the generation of the configuration model, which in the context of


ProductConfiguration.xpt

```
«DEFINE productConfiguration FOR uma::MethodLibrary-»
«FILE "productConfiguration.product" genPath-»

<?xml version="1.0" encoding="UTF-8"?>
<?pde version="3.5"?>

<product «EXPAND productName FOR this-» «EXPAND productID FOR this-» «EXPAND productApplication F

    «EXPAND aboutInfo FOR this-»

    <configIni use="default"></configIni>

    <launcherArgs>
        <programArgs>-perspective moskitt4me.projectmanager.core.perspective</programArgs>
        <vmArgs>-Xms128m -Xmx768m -XX:PermSize=128m -XX:MaxPermSize=512m</vmArgs>
        <vmArgsMac>-XstartOnFirstThread -Dorg.eclipse.swt.internal.carbon.smallFonts</vmArgsMac>
    </launcherArgs>

    «EXPAND windowImages FOR this-»

    «EXPAND splash FOR this-»

    <features>

        «EXPAND fixedFeatures FOR this-»

        «FOREACH technicalFragmentFeatures() AS feature-»
            <feature id="«feature»" version="1.0.0.qualifier"/>
        «ENDFOREACH-»

    </features>
</product>

«ENDFILE»
«ENDDEFINE»
```

FIGURE 4.9: Generation of a product configuration file

Eclipse is represented by a textual *product configuration file* (see Section 2.3.1). Specifically, the excerpt includes the definition of a Xpand rule that is called *productConfiguration*, which produces an XML document entitled *productConfiguration.product*. After the execution of the template, the *.product* file contains the properties of the Eclipse-based environment to be generated during the method implementation phase. Examples of these properties (which can be manually specified by the method engineer or automatically generated) are the name of the environment, the identifier, the about information, the window images, and the splash screen. These properties are included in the product configuration file by means of the *productName*, *productId*, *aboutInfo*, *windowImages*, and *splash* Xpand rules. Of special relevance in the Xpand template is the information that is highlighted in a red rounded rectangle. This part of the template contains the


Feature
Plug-ins
Manifest
Technical Fragment
Product Configuration File
UML2DB_Transformation.ras.zip
Nombre
productConfiguration.product
Dependencies
Features

..
es.cv.gvcase.linkers.uml2db.feature
es.cv.gvcase.linkers.uml2db.sync
es.cv.gvcase.linkers.uml2db.transf
rasset.xml

List the features that constitute the product. Nested features need not be listed.

es.cv.gvcase.linkers.uml2db.feature (1.0.0.qualifier)

Figure 4.10: Example of generation of Eclipse features

code that generates the Eclipse features (which represent the *tools* of the metamodel of Figure 4.7) to be deployed in the CASE environment. Specifically, the rule *fixedFeatures* is in charge of the generation of the features that represent static tools (i.e., tools that are always included in the CASE environment): the process engine. On the other hand, the *foreach* loop generates the Eclipse features that represent dynamic tools (i.e., tools that depend on the method that is being supported). In each iteration, the loop generates an XML element of type <*feature*>. Each of these XML elements represents a technical fragment that is referenced in the method model. These technical fragments are identified by means of the *technicalFragmentFeatures()* operation, which explores the method model by means of an algorithm that is implemented in Java.

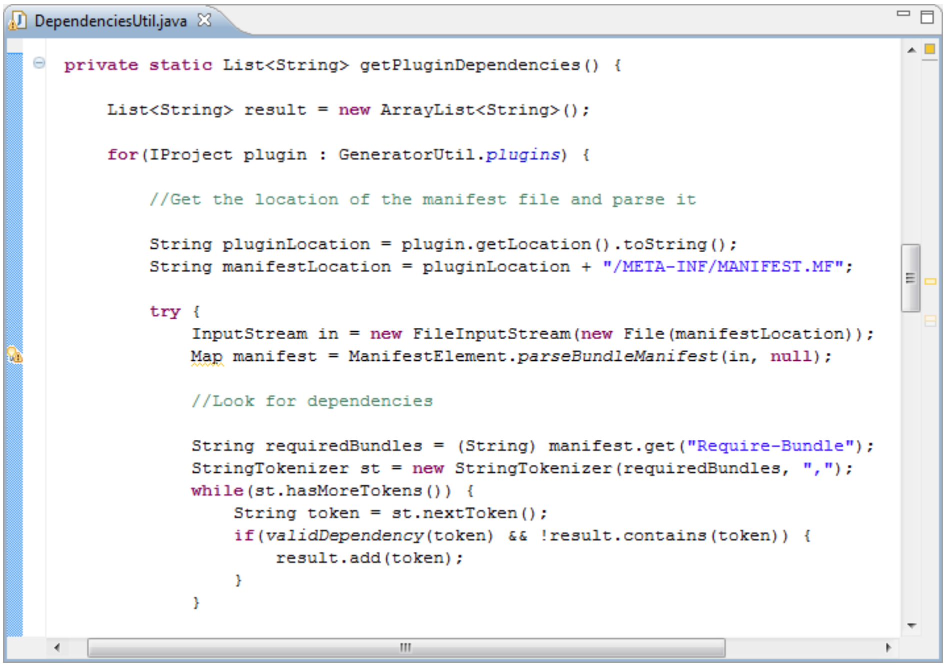
Figure 4.10 graphically illustrates how the Eclipse features are included in the product configuration file. As the figure shows, the technical fragments not only contain Eclipse plug-ins but also an Eclipse feature that groups all of these plug-ins. The identifier of this feature is extracted from the fragment (i.e., the *zip* file) and stored in the product configuration file.

Two final remarks about the *Identification of Software Tools* step are the following. First, note that the CASE environment will not incorporate support for those tasks that do not have any technical fragment associated to them or their output products. These tasks are assumed to be manual tasks, which must be performed by software engineers outside of the context of the software environment. Second, it is also important to note that the identifiers of the technical fragments of type "external tool" must not be included in the configuration model since these tools cannot be deployed in the final software environment.

4.3.1.2 Resolution of Dependencies

The second step of the process involves the resolution of dependencies of the software tools that are included in the configuration model. In other words, the second step involves the identification of all the software tools that must be deployed in the final CASE environment so that the tools that are identified in the previous step can properly function. The identification of software dependencies is enabled by the dependency relationships that connect the technical fragments of the repository (see Section 4.2.2.1) since the software tools of the configuration model are stored in these technical fragments. In order to identify the required software dependencies (which must also be included in the configuration model) the model transformation iterates – in the configuration model – the tools that were identified in the previous step. For each of these tools, the model transformation accesses the corresponding technical fragment in order to extract its software dependencies. The software dependencies of a technical fragment can be calculated by recursively exploring its associated dependency tree (see Section 4.2.2.1). When the software dependencies of all the software tools are included in the configuration model, this model can be considered to be complete.

In order to illustrate the *Resolution of Dependencies* step, Figure 4.11 shows an excerpt of the Java code that implements this step in MOSKitt4ME. Prior to the description of the Java code, it is important to note that, in the context of Eclipse, the dependencies of a software tool are specified within the Eclipse plug-ins that implement the tool (specifically, in a file that is called *manifest.mf*); for this reason, the algorithm that is shown in Figure 4.11 accesses this file, rather than the dependency trees that are stored in the repository. Having introduced this issue, we can explain the excerpt of Java code, which, as Figure 4.11 shows, corresponds to a Java method that is called *getPluginDependencies*. This method iterates a variable called *plugins*, which contains all of the Eclipse plug-ins of the technical fragments that are referenced in the configuration model. For each plug-in, the algorithm gets the location of the *manifest.mf* file; then, the algorithm parses the file into a variable called *manifest*. This variable is used to access the *require-bundle* attribute, which contains all of the dependencies of the Eclipse plug-in. These dependencies are added to a collection called *result*, which is filled iteratively until all of the required dependencies have been identified. Finally, the *result* variable is returned by the Java method so that all of the dependencies that



```
private static List<String> getPluginDependencies() {

    List<String> result = new ArrayList<String>();

    for(IProject plugin : GeneratorUtil.plugins) {

        //Get the location of the manifest file and parse it

        String pluginLocation = plugin.getLocation().toString();
        String manifestLocation = pluginLocation + "/META-INF/MANIFEST.MF";

        try {
            InputStream in = new FileInputStream(new File(manifestLocation));
            Map manifest = ManifestElement.parseBundleManifest(in, null);

            //Look for dependencies

            String requiredBundles = (String) manifest.get("Require-Bundle");
            StringTokenizer st = new StringTokenizer(requiredBundles, ",");
            while(st.hasMoreTokens()) {
                String token = st.nextToken();
                if(validDependency(token) && !result.contains(token)) {
                    result.add(token);
                }
            }
```

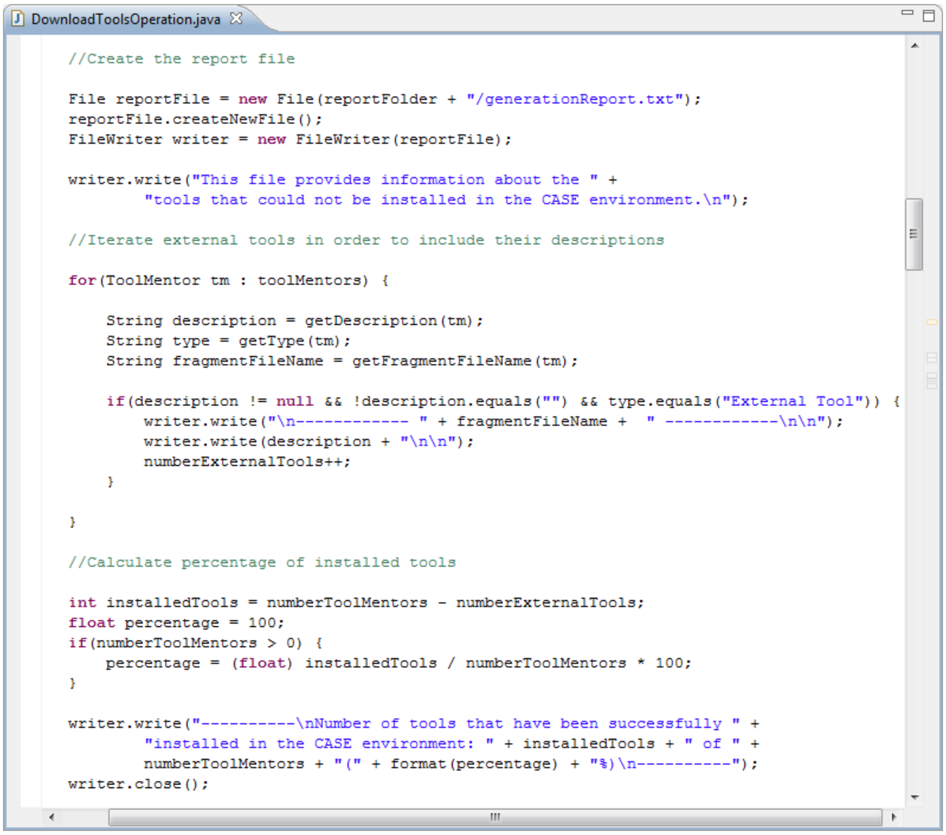
FIGURE 4.11: Resolution of software dependencies

have been identified can be added to the product configuration file by means of a Xpand rule – this is not shown in the figure.

4.3.1.3 Deployment of Software Tools

The last step of the process involves the deployment of the tools that are included in the configuration model into the final software environment. The first tool to be deployed is the process engine. Prior to the deployment of this tool, the BPMN 2.0 model that is obtained in the *Executable Process Generation* step (see Section 4.2.3.3) is deployed into the engine database. Thus, the process engine can execute instances of the process part of the method. Once the process engine is fully integrated in the CASE environment, this environment is completed with the software tools that are included in the configuration model.

As a result of the generation process, our model transformation also produces a *generation report*. This report contains information about the number of tools



```
//Create the report file

File reportFile = new File(reportFolder + "/generationReport.txt");
reportFile.createNewFile();
FileWriter writer = new FileWriter(reportFile);

writer.write("This file provides information about the " +
        "tools that could not be installed in the CASE environment.\n");

//Iterate external tools in order to include their descriptions

for(ToolMentor tm : toolMentors) {

    String description = getDescription(tm);
    String type = getType(tm);
    String fragmentFileName = getFragmentFileName(tm);

    if(description != null && !description.equals("") && type.equals("External Tool")) {
        writer.write("\n------------ " + fragmentFileName +  " ------------\n\n");
        writer.write(description + "\n\n");
        numberExternalTools++;
    }

}

//Calculate percentage of installed tools

int installedTools = numberToolMentors - numberExternalTools;
float percentage = 100;
if(numberToolMentors > 0) {
    percentage = (float) installedTools / numberToolMentors * 100;
}

writer.write("----------\nNumber of tools that have been successfully " +
        "installed in the CASE environment: " + installedTools + " of " +
        numberToolMentors + "(" + format(percentage) + "%)\n----------");
writer.close();
```

FIGURE 4.12: Automatic production of the the generation report

that have been successfully installed in the CASE environment and how software engineers must proceed to obtain full software support for the method. In general, all of the software tools that are not defined as "external tool" can be integrated in the CASE environment. Therefore, the generation report contains textual information about the external tools. This information is extracted from the attributes of the technical fragments (see Section 4.2.2.2).

In order to illustrate the automatic production of the generation report, Figure 4.12 shows an excerpt of the Java code that produces this report in MOSKitt4ME. Specifically, the excerpt shows how the textual file is created first. Then, all of the technical fragments that are installed in the CASE environment are iterated. For each of these fragments, the algorithm retrieves three properties: description, type, and name. The *type* property is used to distinguish external tools from other types of fragments. For each external tool, the algorithm prints the *name* of the tool and also its *description*. Once all of the technical fragments have been iterated, the algorithm calculates the percentage of tools that have been successfully deployed in the CASE environment.

Note that, in this section, we do not illustrate the implementation of the *Deployment of Software Tools* step. This is because this step is fully implemented by the Eclipse PDE, and, therefore, it has not been necessary to implement the deployment of software tools in MOSKitt4ME. Specifically, the Eclipse PDE allows users to automatically obtain fully functional Eclipse RCP products directly from the specification of these products in product configuration files.

4.4 Method Execution

The method execution phase involves the enactment of method instances using the software environment that is obtained in the method implementation phase. The method execution is, therefore, tool-assisted; that is, it is guided by an integrated environment that provides assistance for software engineers during the entire process of software development. To provide this assistance, the software environment comprises various components: the *Method Process Support* (which includes an engine that enables the execution of the method process part) and the *Method Product Support* (which includes a set of tools that enable the creation of the method products); additionally, the software environment incorporates a component that offers an intuitive graphical user interface for end users. We call this component the *Project Manager*. These components (i.e., the project manager, the method process support, and the method product support) are graphically depicted in Figure 4.13 and detailed below.

4.4.1 The Project Manager Component

The *Project Manager* component provides a graphical user interface for the CASE environment and assists software engineers during the method execution. To do this, the project manager uses the process engine to execute BPMN 2.0 process

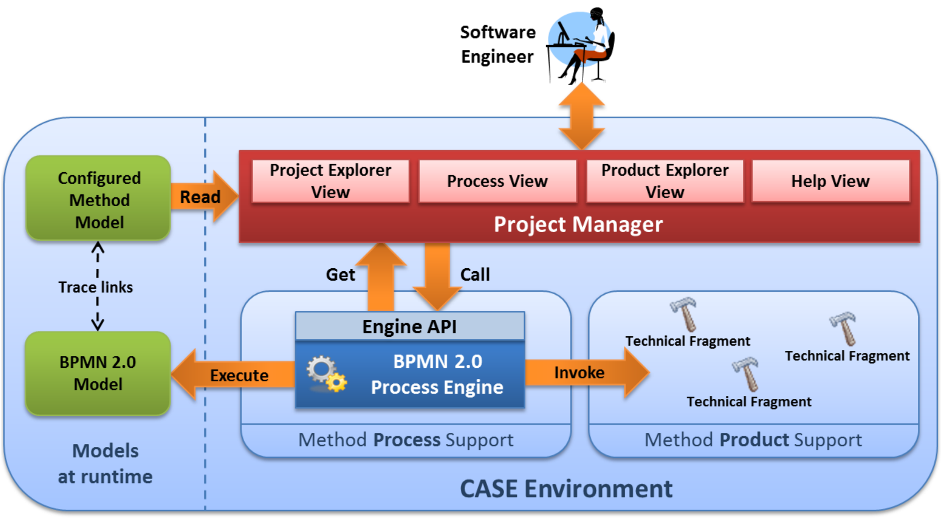


FIGURE 4.13: The method execution phase

instances; additionally, the project manager makes use of the method model at runtime to extract information about the method that is not represented in the BPMN 2.0 model.

The project manager is a static component; that is, it is always included in the CASE environment regardless of the method that is being supported. It is divided into four views – the project explorer, the process view, the product explorer, and the help view – which provide software engineers with information of different nature. These four views are detailed below.

Project Explorer. The *project explorer* view provides a tree-based representation of the workspace. This view is hierarchically organized in projects, folders, and files. From the *project explorer*, software engineers can create new projects, delete existing projects, add files/folders to these projects, etc. This kind of functionality is implemented, for example, by the Eclipse Package Explorer (which is provided as part of the JDT), the Resource Explorer of MOSKitt, or the Eclipse Project Explorer.

Process. The *process* view shows the current state of the process instance that is associated to the project that is selected in the *project explorer*. From the *process* view, software engineers can invoke the execution of the tasks that are executable as well as determine which tasks are non-executable or have already been executed. Once a task is finished, the Project Manager invokes the API of the BPMN 2.0 Process Engine to set the task as executed and proceed to the next state of the process. The *process* view also enables task filtering based on the role of the users; thus, the software engineer may focus on the tasks that he/she is in charge of their performance. Another type of task filtering is based on the task state; thus, software engineers may, for instance, focus on the tasks that are executable in the current state of the project.

Product Explorer. The *product explorer* view shows a hierarchical picture of the artifacts that have been produced during the course of the project that is selected in the *project explorer* view. To illustrate how this hierarchy is organized, let us consider a file that is called "classModel.uml", which represents a specific UML class diagram. The *product explorer* may show this file under a product called "UML 2.0 model", which represents a product that is defined in the method model. The semantics of this nesting relationship is that the file represents an instantiation of the product in a specific software development project. Additionally, the product may, in turn, be nested in a domain called "Models", which represents a product category that is also defined in the method model. Thus, the *product explorer* shows a hierarchical representation of artifacts that is based on domains, subdomains, and product elements, all of which are read from the method model. On the other hand, the *product explorer* view also enables product filtering based on the role of the user; thus, the software engineer may focus on those artifacts that he/she is responsible for.

Help. The *Help* view provides guidance for software engineers during the performance of the method tasks. This view is dynamically updated based on the task that is selected in the *process* view. The guidelines of a specific task are known by the *Help* view because they are associated to the task as guidance fragments (see Section 4.2.2.1). One example of this type of view is the Eclipse Help view, which is provided by the Help System that is integrated in Eclipse (see Section 2.3.1).

4.4.2 Method Process Support

The process engine is the component that provides process support in the software environment that is obtained in the method implementation phase. Similarly to the project manager, the engine is a static component: it is always included in the environment regardless of the method that is being supported. As Figure 4.13 illustrates, the process engine is the central component of the software environment. By using the BPMN 2.0 model at runtime, the process engine provides up-to-date and exact information about the running process instances, and, thus, it drives the behavior of the software environment. Driving the behavior of the software environment refers to the fact that the project manager (by communicating with the process engine) restricts its functionality according to the state of the process instances; for example, the user can only create a limited set of products (specifically, the output products of the executable tasks). Additionally, the software environment will behave differently depending on the type of task that is being executed by the process engine:

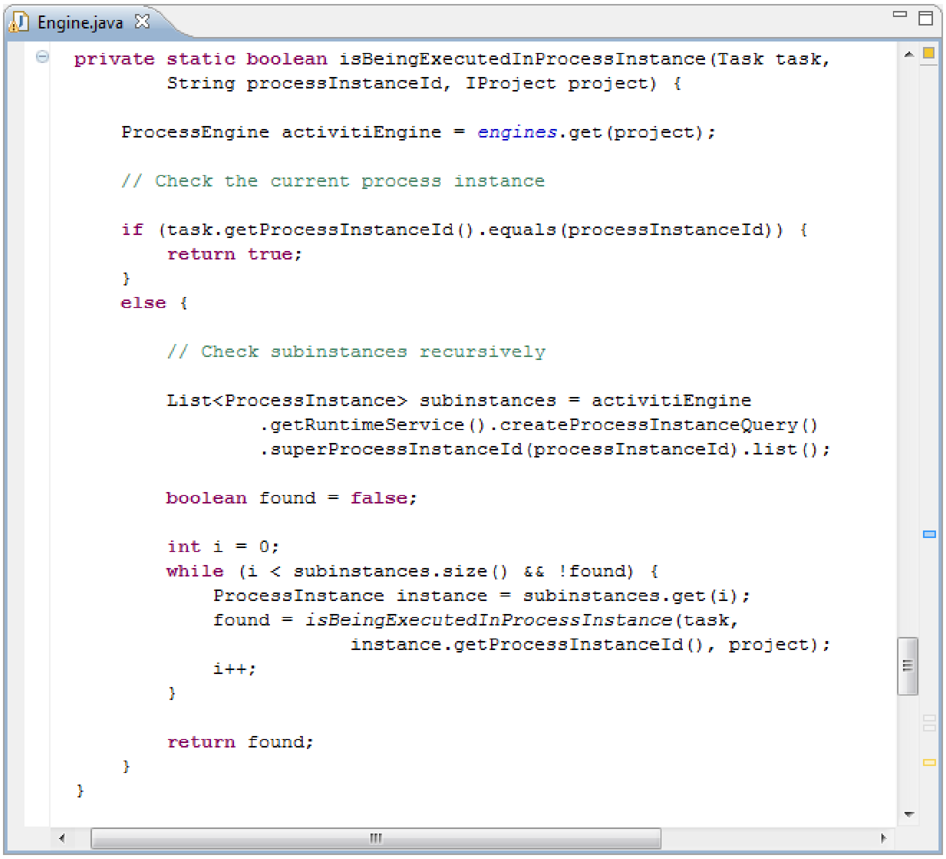
- *Service task:* when a service task becomes active, the process engine starts the task execution. The execution of a service task involves the invocation of the model transformation that is associated to the task as a technical fragment. When a model transformation is invoked, the software environment opens a dialog that (optionally) allows the user to specify a configuration model [172] as well as to cancel the execution of the transformation. When the model transformation finishes, the software environment shows a popup message that summarizes the transformation results.
- *User task:* when a user task becomes active, the process engine invokes the software tools that are associated to the output products of the task as technical fragments. When these tools are invoked, the software environment opens the creation wizards that allow software engineers to create the products (e.g., empty models or textual files); then, the environment opens the tools that enable the editing of these products (e.g., graphical or textual editors).
- *Manual task:* when a manual task becomes active, neither the process engine nor the software environment performs any action since this type of tasks are performed without the aid of any software tool. Thus, the control of the

process is transferred to the software engineer, who must manually indicate when the task is finished so that the process can proceed to its next state.

- *Call activity:* when a call activity becomes active, the process engine automatically starts a new instance of the BPMN 2.0 process that is referenced by the call activity. This action simply involves a refresh of the *process* view of the Project Manager component so that it shows an updated version of the running process instance.

In order to offer all of the above functionality, the software environment (more specifically, the Project Manager component) must communicate with the process engine through its API. This communication is represented in Figure 4.13 by the *Get* and *Call* arrows. By communicating with the process engine, the Project Manager can perform actions such as deleting running process instances (when development projects finish), creating new process instances (when new projects start or call activities are executed), or setting tasks as executed (when software engineers conclude the tasks). Due to the high frequency that the actions involving the use of the process engine have at the code level, we implemented a *facade design pattern* [173, 174] to facilitate the access to the API of the process engine. Our facade is a Java class that provides a unified interface to the set of inferfaces that comprise the engine API. This unified interface offers a set of convenient methods that implement commonly recurring tasks that involve the use of the process engine.

In order to illustrate our unified interface, Figure 4.14 shows one of the Java methods that the interface provides. As the figure shows, the Java method is called *isBeingExecutedInProcessInstance*. This method is in charge of checking whether or not a task is executable (at the moment of invocation of the method) in the context of a specific process instance (considering also the subinstances that were created by means of the execution of call activities). To achieve this goal, the Java method checks the current process instance first. To this end, the API of the process engine is used. Then, the method checks the subinstances recursively. To obtain the subinstances of the current process instance, the method creates a *process instance query*, which is provided by the *runtime service* of the process engine API.



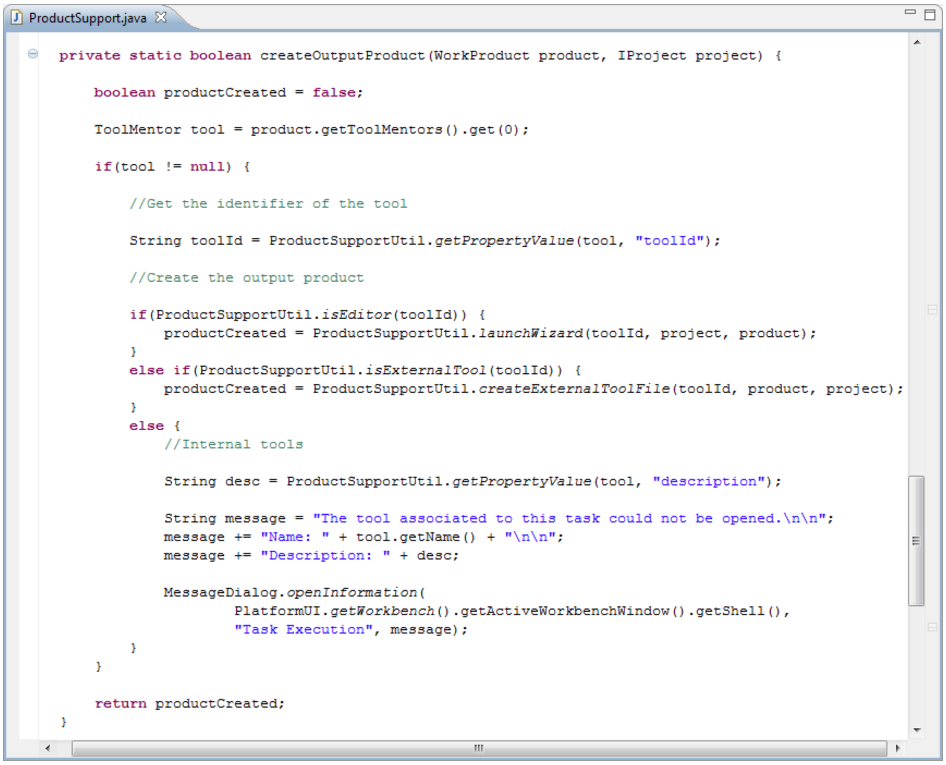
```java
private static boolean isBeingExecutedInProcessInstance(Task task,
        String processInstanceId, IProject project) {

    ProcessEngine activitiEngine = engines.get(project);

    // Check the current process instance

    if (task.getProcessInstanceId().equals(processInstanceId)) {
        return true;
    }
    else {

        // Check subinstances recursively

        List<ProcessInstance> subinstances = activitiEngine
                .getRuntimeService().createProcessInstanceQuery()
                .superProcessInstanceId(processInstanceId).list();

        boolean found = false;

        int i = 0;
        while (i < subinstances.size() && !found) {
            ProcessInstance instance = subinstances.get(i);
            found = isBeingExecutedInProcessInstance(task,
                        instance.getProcessInstanceId(), project);
            i++;
        }

        return found;
    }
}
```

FIGURE 4.14: Example of Java method using the Activiti Engine API

4.4.3 Method Product Support

The software tools that are encapsulated in technical fragments are the components that provide product support in the software environment that is obtained in the method implementation phase. Unlike the project manager and the process engine, the technical fragments are dynamic components: their inclusion in the software environment depends on the method that is being supported. The technical fragments that are included in the environment correspond to those that are referenced in the method model (with the exception of the external tools).

In a similar way to the process engine, the technical fragments also play an important role in determining how the environment behaves. This is because the


ProductSupport.java

```java
private static boolean createOutputProduct(WorkProduct product, IProject project) {

    boolean productCreated = false;

    ToolMentor tool = product.getToolMentors().get(0);

    if(tool != null) {

        //Get the identifier of the tool

        String toolId = ProductSupportUtil.getPropertyValue(tool, "toolId");

        //Create the output product

        if(ProductSupportUtil.isEditor(toolId)) {
            productCreated = ProductSupportUtil.launchWizard(toolId, project, product);
        }
        else if(ProductSupportUtil.isExternalTool(toolId)) {
            productCreated = ProductSupportUtil.createExternalToolFile(toolId, product, project);
        }
        else {
            //Internal tools

            String desc = ProductSupportUtil.getPropertyValue(tool, "description");

            String message = "The tool associated to this task could not be opened.\n\n";
            message += "Name: " + tool.getName() + "\n\n";
            message += "Description: " + desc;

            MessageDialog.openInformation(
                    PlatformUI.getWorkbench().getActiveWorkbenchWindow().getShell(),
                    "Task Execution", message);
        }
    }

    return productCreated;
}
```

Figure 4.15: Excerpt of code implementing the method product support

process engine, which is the central driver of the environment behavior, needs to access these fragments in order to function properly. To illustrate this idea, let us consider that a user task is active. As Section 4.4.2 explains, the execution of a user task involves the invocation of the software tools that are associated to the output products of the task as technical fragments. Since the type of these tools is variable (e.g., they can be graphical editors, textual editors, or external tools), the behavior of the software environment will vary depending on the type of the technical fragment that is being invoked. For this reason, the process engine needs to obtain the type of these fragments, and, then, perform different actions based on these types. This is illustrated in the excerpt of Java code that is shown in Figure 4.15.

As Figure 4.15 shows, to create an output product of a user task, the identifier of the technical fragment that is associated to this product is retrieved first. This technical fragment, in the context of the EPF Composer, is represented by an element of type *Tool Mentor*. Once the identifier is retrieved, the algorithm checks the type of the technical fragment. If the fragment encapsulates an editor[3], then the Java method launches the creation wizard of the editor so that the user can create the corresponding product. If the fragment represents an external tool, then the Java method creates a file that is editable by this external tool. For instance, if the external tool is Microsoft Word, a *.doc* file will be created – the information about the required file extension is stored in the technical fragment. Finally, in case the technical fragment represents an internal tool, a message will be shown to the user so that he/she gets instructions on the use of the tool – the content of the message is also obtained from the technical fragment. Note that the process engine cannot open internal tools because, unlike the other types of tools, they do not implement a common interface (see Section 4.2.2.1).

4.5 Conclusions

This chapter introduces a methodological approach that supports model-driven Method Engineering. The introduced approach has various unique characteristics. Most notably, our approach is built upon MDE technology, which has allowed us to properly handle the inherent complexity of Method Engineering. The use of models and model transformations (together with leveraging models at runtime) has made possible the definition of an approach that equally encompasses the design, implementation, and execution of methods, supporting also the method product and process parts. This is unlike existing Method Engineering approaches, which in general focus on the product part of methods and only support one of the phases of the Method Engineering lifecycle. In order to overcome these limitations, our approach defines a DSL that benefits from two widely acknowledged standards: SPEM 2.0 and BPMN 2.0. While the concepts of SPEM 2.0 enable the definition of complete software development methods, the concepts of BPMN 2.0 allow method engineers to enhance method definitions in terms of process specification. On the other hand, since one of the major goals of our approach is to promote reusability, we define a taxonomy of method fragments that enable rapid method

[3]The technical fragments of type "Metamodel" and "Grammar" are considered in this category. In these cases, the CASE environment provides a default editor for the creation of the products.

design by means of their assembly. This taxonomy does not only takes into consideration the conceptual aspects of methods, but it supports the technical aspects as well. This allows method engineers to define methods in a conceptual manner and later obtain customized software environments that support the method execution. Supporting the construction of these software environments is a significant advantage of our approach; it allows method engineers to provide customized tool support for many aspects of Software Engineering (such as the creation of method products or the management of the roles of the development team).

In addition to all of the above characteristics, another distinctive aspect of our methodological approach is the software infrastructure that has been developed to support the approach. This software infrastructure is detailed in Chapter 5.

Chapter 5

MOSKitt4ME: A Software Infrastructure

Models play a pivotal role in our methodological approach for Method Engineering. They allow method engineers to express method designs in terms of high-level concepts, and, since the resulting models are machine-processable and precise enough, they can be used to automate part of the Method Engineering lifecycle. However, in the absence of appropriate tool support, models can become a heavy burden and this burden can be further aggravated by the inherent complexity of Method Engineering. The necessity of software tools for Model-Driven Engineering (MDE) has already been emphasized in some relevant contributions (see for example [43, 111]), while authors such as Bajec *et al.* [6], Niknafs *et al.* [41], and Harmsen [20] highlight the need for appropriate tool support in Method Engineering.

In order to meet this need for software tools, we implemented a Computer-Aided Method Engineering (CAME) environment that provides complete software support for the methodological approach that is presented in Chapter 4. To ensure that the CAME environment offered the necessary functionality, we identified a set of functional requirements prior to developing the tool. Then, after these requirements were identified, we defined the architecture of our CAME environment at a conceptual level from a technology-independent perspective. By defining the architecture in a technology-independent manner, its components are generic enough to enable their implementation in different software platforms. Specifically, in the context of this thesis, we implemented the architecture as an extension of MOSKitt. This implementation resulted in an Eclipse-based CAME environment that is called MOSKitt4ME (which stands for MOSKitt for Method Engineering).

The remainder of the present chapter is structured as follows. First, Section 5.1 describes the set of functional requirements that we identified to facilitate the development of MOSKitt4ME. Then, Section 5.2 introduces the MOSKitt4ME architecture and also how we implemented this architecture in the context of Eclipse. Section 5.3 provides an in-depth description of the functionality that is offered by MOSKitt4ME. Finally, Section 5.4 draws some conclusions about the present chapter.

5.1 Functional Requirements

In order to ensure that we provide adequate software support for our methodological approach, we defined a set of functional requirements that guided the development of MOSKitt4ME. To determine these requirements, we analyzed the functionality that MOSKitt4ME must provide in each of the three phases that comprise our approach: design, implementation, and execution. With respect to the method design phase, we identified three functional requirements (Req. 1, Req. 2, and Req. 3):

- **Req. 1. MOSKitt4ME must enable the conceptual modeling of the product and process parts of software development methods.** In order to meet this requirement, MOSKitt4ME must incorporate an editor that supports the DSL that is defined in our methodological approach (see Chapter 4). This editor will preferably be a graphical editor; thus, it will be able to support the concrete syntax that is defined by SPEM 2.0 and BPMN 2.0. A tree-based editor can also be included in MOSKitt4ME. Unlike these types of editors, a textual editor cannot be included since our DSL does not define a textual syntax. Graphical and tree-based editors (when compared to textual editors) have the advantage that they offer higher level visual constructs, which are more intuitive and easier to use and learn.

- **Req. 2. MOSKtit4ME must allow users to build executable representations of the method models.** Our DSL does not define execution semantics. Therefore, to meet this requirement, MOSKitt4ME must provide a model transformation that obtains executable models from the methods that are defined using our DSL; thus, the software environments that support the methods will be able to support method execution.

- **Req. 3. MOSKitt4ME must support the creation, storage, retrieval, and integration of conceptual and technical fragments.** In order to meet this requirement, MOSKitt4ME must allow method engineers to connect to repositories of conceptual and technical fragments. This can be accomplished by means of a repository client. This client, when connected to a repository, must allow users to store new fragments and to search/retrieve existing fragments according to their properties. For the creation of conceptual fragments, the client must allow method engineers to select parts of the method under construction and to encapsulate these parts in reusable assets; the repository client must also enable the integration of conceptual fragments into the method that is being defined by method engineers. For the creation of technical fragments, MOSKitt4ME must incorporate metatools that facilitate the construction of tools for software development. These tools must not be limited to graphical editors, but other types of tools (e.g., textual editors or model transformations) must be supported as well. The encapsulation of these tools in reusable assets must also be supported by the repository client, which, in turn, must enable the linkage between these assets and the method elements.

In addition to the above requirements, we also identified one functional requirement (Req. 4) that relates to the method implementation phase:

- **Req. 4. MOSKitt4ME must enable the automatic generation of software environments that support the methods.** To meet this requirement, MOSKitt4ME must incorporate a software component that implements the process that is described in Section 4.3.1. To implement this process, the software component must be able to interpret the method models that are created by means of the method editor (see Req. 1). Additionally, the component must be able to obtain information from the technical fragments (e.g., their software dependencies). This can be done by enabling the communication between the component and the repository client (see Req. 3).

Finally, we also identified one functional requirement (Req. 5) that relates to the method execution phase:

- **Req. 5. The software environments that are automatically generated by MOSKitt4ME must incorporate a set of integrated tools that support the product and process parts of methods.** In order to provide product support, the software environments must enable (by means of tools such as graphical editors or code generators) the creation and manipulation of the artifacts that are defined in the methods. The process support must be provided in the form of a process engine, which will offer enactment facilities (such as task orchestration) that enable the execution of the method process part. In addition to these tools, the software environment will also provide a graphical user interface that assists software engineers during the entire process of software development.

5.2 Developing MOSKitt4ME

After the identification of the functional requirements that are introduced in Section 5.1, we developed the MOSKitt4ME environment. To carry out this development, we defined the architecture of MOSKitt4ME first; then, we implemented the architecture in a specific software platform: MOSKitt. These two steps are detailed in Sections 5.2.1 and 5.2.2, respectively.

5.2.1 The Architecture of MOSKitt4ME

The architecture of MOSKitt4ME has been specifically designed to meet the requirements that are described in Section 5.1. For the definition of the architecture,

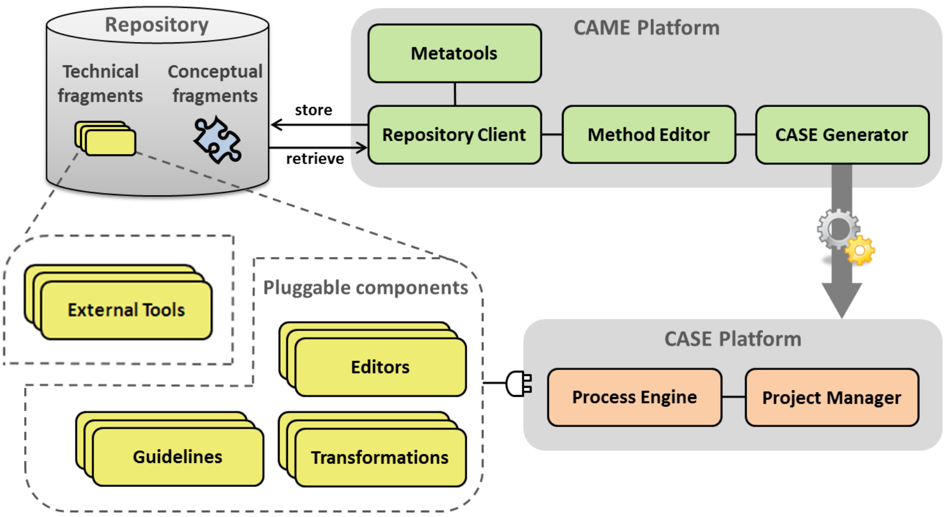


FIGURE 5.1: A software architecture for Method Engineering

we rely on the *component* concept since this is a well-understood concept that can be implemented in most of the implementation technologies available. Components can be defined as the basic software pieces that conform the architecture of a system.

Figure 5.1 shows the technology-independent components that conform the architecture of MOSKitt4ME. As the figure shows, the architecture is composed of two main parts: the *CAME platform* and the *CASE platform*. The CAME platform is composed of four components: *Method Editor*, *Repository Client*, *Metatools*, and *CASE Generator*. The CASE platform is composed of a *Process Engine* and a *Project Manager*; the functionality of the CASE platform can also be extended via *Pluggable Components* and *External Tools*, which are stored in a repository as technical fragments. Below, we provide more details about all of these components.

Method Editor. The method editor allows method engineers to perform the conceptual modeling of methods according to our DSL. The inclusion of this component in the software architecture of MOSKitt4ME fulfills Req. 1. In order to fulfill Req. 2, the method editor must incorporate a model transformation that obtains executable representations of the method models.

Repository Client. The repository client allows method engineers to connect to repositories of conceptual and technical fragments. By means of this client, method engineers can create new fragments and also store, retrieve, and integrate them. Thus, the inclusion of the repository client in MOSKitt4ME fulfills Req. 3.

Metatools. The metatools allow method engineers to develop tools for software development. Examples of metatools are frameworks for the construction of graphical editors or languages for the implementation of code generators. The inclusion of metatools in MOSKitt4ME relates to Req. 3 since they allow method engineers to create new tools and to encapsulate them in technical fragments by means of the repository client.

CASE Generator. The CASE generator is in charge of the automatic (or semi-automatic) construction of the CASE platform, which provides software support to the method that is defined by means of the CAME platform. Thus, the CASE generator, which allows us to fulfill Req. 4, bridges the gap between the conceptual design of the method and its final technical implementation. To achieve this goal, the CASE generator uses the linkage between the conceptual and technical aspects of the method, which is established by means of the repository client. The functionality that is provided by the CASE part of MOSKitt4ME allows us to fulfill Req. 5.

Process Engine. The process engine supports the execution of the process part of the method that was defined using the method editor.

Project Manager. The project manager component provides a graphical user interface that assists software engineers during the course of software development projects. Specifically, this interface gives information regarding the state of the projects and running process instances.

Pluggable Components. The pluggable components represent software tools that can be composed in a pluggable manner to extend the functionality that is offered by the CASE platform. These components can be *editors*, *transformations*, and *guidelines*. Editors allow software engineers to create method products by means of textual or graphical languages. Transformations represent tools that can transform one software artifact into another, for instance a machine-processable model into executable code. Guidelines provide guidance (e.g., textual documentation or process models) that can assist software engineers during the performance of the method tasks. All of these types of tools are (semi)automatically plugged into the CASE platform by means of the CASE generator.

External Tools. The external tools represent software pieces that can extend the functionality that is offered by the CASE platform. Unlike the pluggable components, the external tools cannot be plugged into the CASE platform because they are implemented in a different technology. Nonetheless, the CASE platform contains references to these tools so that they can be opened outside of the environment context in order to enable the creation of specific method products.

5.2.2 Implementation of the Architecture

In this section, we provide implementation details about each of the software components of the architecture that is presented in Section 5.2.1. Specifically, we focus on the software technologies that were used to carry out this implementation. These technologies are graphically depicted in Figure 5.2.

Method Editor. In order to implement this component of MOSKitt4ME, we integrated two editors that have been developed by the Eclipse community: the EPF Composer – a SPEM 2.0 editor that is distributed as part of the Eclipse Process Framework Project – and the Activiti Designer – a BPMN 2.0 editor that is provided as part of the Activiti Project. To integrate these two editors (and the metamodels that define the abstract syntax of their DSLs), we performed two steps. First, we extended the metamodel of the EPF Composer in order to fit the metamodel of our DSL (see Figure 4.2); to this end, we extended the metamodel of the EPF Composer with references to the metamodel of the Activiti Designer. Second, we enhanced the functionality of the EPF Composer so that this editor supports BPMN 2.0; for instance, we added a menu item that allows method engineers to open BPMN 2.0 processes (which are edited by means of the Activiti Designer) within the context of the EPF Composer. In addition to these two steps, we also implemented a M2M transformation that obtains executable BPMN 2.0 models from the method models that are defined using the EPF Composer.

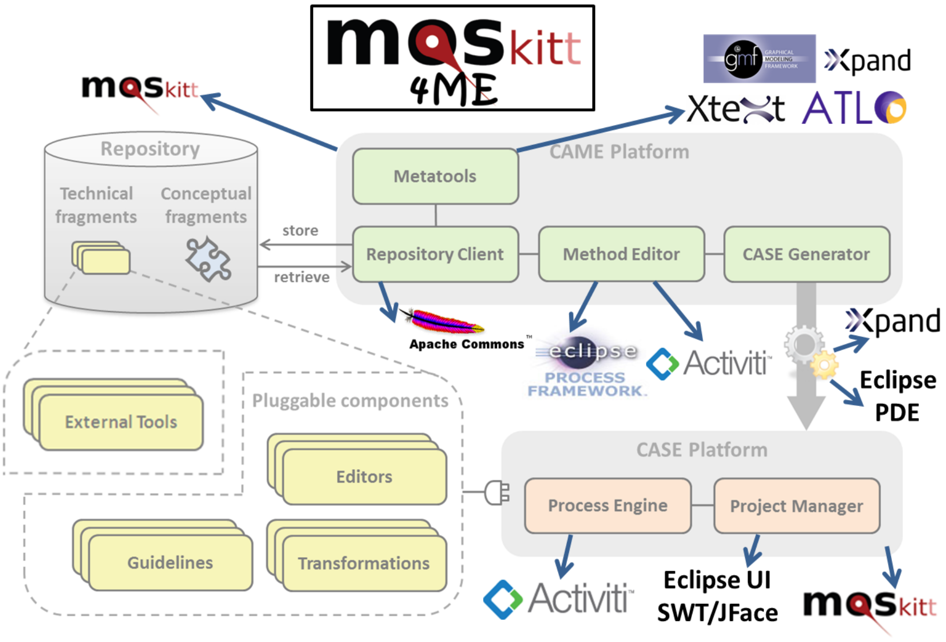


FIGURE 5.2: Technologies used to implement our software architecture

Repository Client. This component has been implemented as an Eclipse view that allows method engineers to connect to FTP repositories. The connection with these repositories is established by means of a FTP client that is provided by the Apache Commons Project[1].

Metatools. XText and GMF allow method engineers to develop textual and graphical editors, respectively. The development of form-based editors is supported by the FEFEM framework that is provided by MOSKitt, while EMF supports the specification of metamodels and the generation of tree-based editors. On the other hand, Model-to-Model (M2M) transformations can be implemented by means of ATL, while Xpand supports the development of code generators; that is, Model-to-Text (M2T) transformations.

CASE Generator. The CASE generator has been implemented in Xpand as a M2T transformation. This transformation obtains an Eclipse product configuration file (which represents the configuration model of Figures 4.6 and

[1] http://commons.apache.org/

4.8) from the method model that is defined by means of the EPF Composer. The configuration file, thanks to the functionality that is offered by the Eclipse PDE, enables the automatic construction of the CASE part of MOSKitt4ME. This CASE part is a reconfiguration of MOSKitt that only contains the Eclipse plug-ins that are strictly required to support the method.

Process Engine. The process engine of MOSKitt4ME is the Activiti Engine, which is a Java process engine that runs BPMN 2.0 processes and is provided as part of the Activiti project. The Activiti Engine can run the BPMN 2.0 processes that are built by means of the Activiti Designer.

Project Manager. The project manager component has been implemented in Java as a set of Eclipse views. For the implementation of these views, we used the functionality that is provided by the Eclipse community to develop graphical user interfaces, mainly the plug-ins of the Eclipse UI packages and the SWT/JFace libraries.

Pluggable Components. These components can be developed using any technology as long as they are implemented as Eclipse plug-ins. Nonetheless, using the metatools of MOSKitt4ME enables the subsequent communication between the pluggable components and the project manager.

External Tools. These tools can be implemented in any technology; they are not part of MOSKitt4ME but rather installed in the system. MOSKitt4ME contains references to the external tools so that they can be opened when necessary. These tools can be opened within MOSKitt4ME since Eclipse provides mechanisms for running tools that are not part of the platform.

5.3 The MOSKitt4ME Environment

By leveraging the technologies that are described in Section 5.2.2, we developed the MOSKitt4ME environment. MOSKitt4ME has been conceived as an extension of MOSKitt; for this reason, the installation of MOSKitt4ME must be performed in two steps. The first step involves the download and installation of MOSKitt 1.3.10 from the MOSKitt website (http://www.moskitt.org/). Then, in the second step, MOSKitt4ME has to be installed within MOSKitt by means of the following update site: http://www.pros.upv.es/moskitt4me/Updates. To install MOSKitt4ME from this update site, the user must select "Install New Software..." from the "Help" menu; then, push the "Add..." button to add the MOSKitt4ME update site; and, finally, follow the wizard to install the tool. It is important to activate the checkbox "Contact all update sites during install to find required software" so that all of the MOSKitt4ME dependencies can be resolved during the installation.

In the following sections, we detail the functionality of MOSKitt4ME. We focus on the CAME part of MOSKitt4ME first, and, then, we detail the functionality of the CASE part. Further information about the functionality of MOSKitt4ME is available in the user guide, which can be downloaded from the MOSKitt4ME website: `http://users.dsic.upv.es/~mcervera/moskitt4me`.

5.3.1 The CAME Part of MOSKitt4ME

When MOSKitt4ME is successfully installed, the CAME part of the tool is added to MOSKitt; note that the CASE part is not included because a separate environment will be generated during the phase of method implementation. As Section 5.2.1 describes, the CAME part of MOSKitt4ME is composed of the following software components: method editor, repository client, metatools, and CASE generator. These components are detailed in Sections 5.3.1.1, 5.3.1.2, 5.3.1.3, and 5.3.1.4, respectively.

5.3.1.1 Specifying Methods: the Method Editor

The method editor of MOSKitt4ME can be accessed by opening the "Method Design" perspective. This perspective can be opened by means of the "Open Perspective" dialog, which is available at the "Window" menu. When the "Method Design" perspective is opened, the MOSKitt4ME workbench is organized in three different parts, which are graphically depicted in Figure 5.3. These three parts are the following:

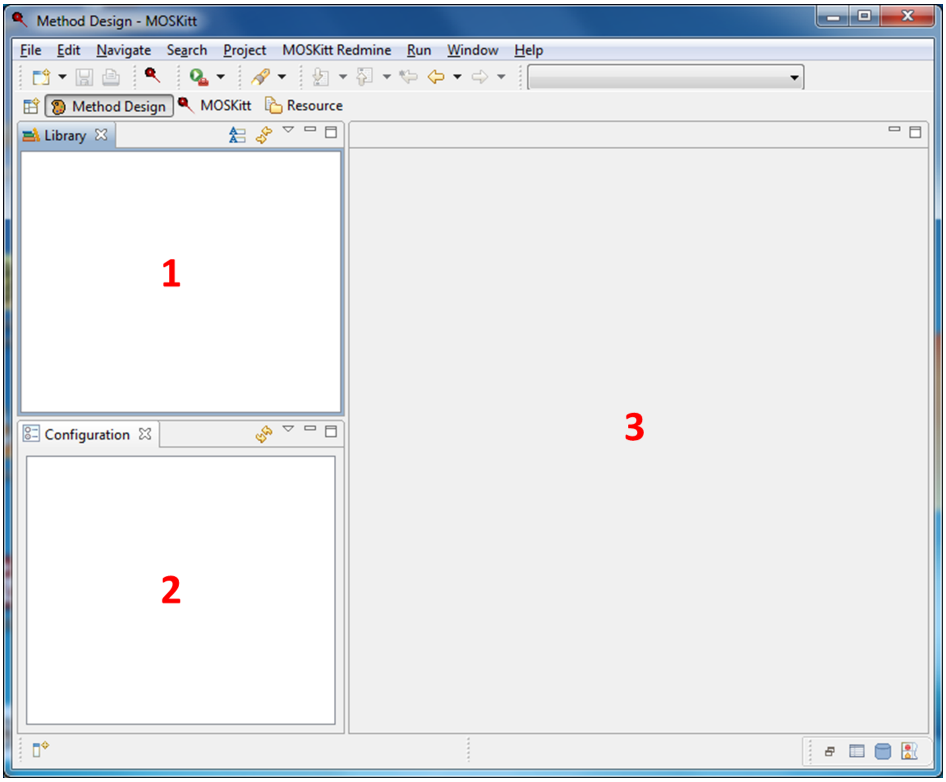


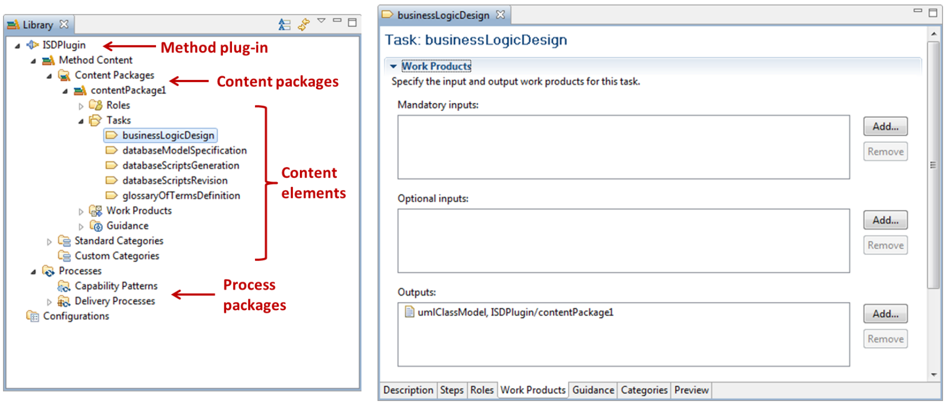
FIGURE 5.3: Parts of the method design perspective

1. *Library view:* this view offers a hierarchical picture of the elements that compose the method that is being defined by method engineers. These elements conform to the SPEM 2.0 standard; some examples are tasks, roles, and work products.

2. *Configuration view:* this view shows a subset of the elements that are displayed in the Library view; thus, it provides a mechanism for defining partial views on methods.

3. *Editor area:* the properties of the method elements that are selected in the Library view can be edited by means of specific editors that are opened by default on the right side of the MOSKitt4ME workbench.

Both the Library view and the Configuration view of MOSKitt4ME are provided by the EPF Composer. Below, we provide a brief summary of how methods are



Task: businessLogicDesign

Work Products

Specify the input and output work products for this task.

Mandatory inputs:

Optional inputs:

Outputs:

umlClassModel, ISDPlugin/contentPackage1

FIGURE 5.4: The library view

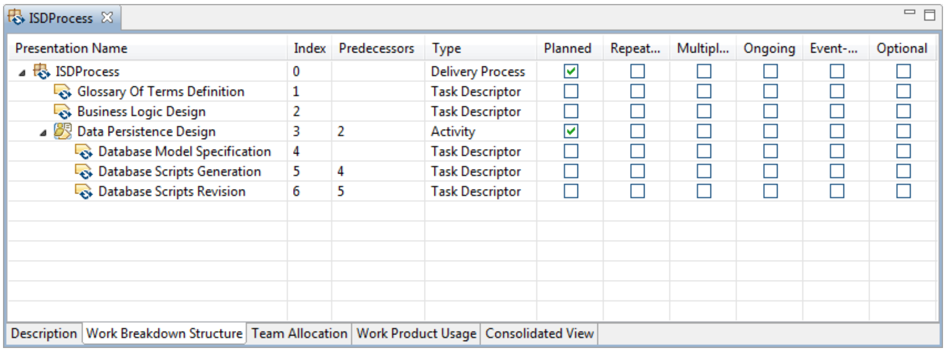
defined by means of this editor. For further details, we refer the reader to the user guide that is provided in the Eclipse website [175].

EPF Composer

The definition of methods by means of the EPF composer is carried out in two steps. First, method engineers define the method content; second, the method content is instantiated in specific development processes (see Section 2.1.2.1). Both method content and processes are defined within method libraries, which can be created by means of the "File" menu: "New Method Library". When a method library is created, it can be edited by means of the Library view. The Library view can only show one method library at a time.

In a method library, method content and processes are organized in hierarchical structures named method plug-ins. Method plug-ins contain content packages, which define method content, and process packages, which define processes. This hierarchy is graphically depicted in Figure 5.4 (left).

Within content packages, new elements can be added and existing elements can be deleted by means of the contextual menu. The properties of the content elements can be specified by means of form-based editors that are opened in the editor area by selecting "Edit" on the contextual menu (or double-clicking the content elements). Figure 5.4 (right) shows an example of these form-based editors. Specifically, in the example, the editor allows the user to specify the inputs and outputs of the task that is selected in the Library view. In this case, the task,



Presentation Name	Index	Predecessors	Type	Planned	Repeat...	Multipl...	Ongoing	Event-...	Optional
ISDProcess	0		Delivery Process	✓	☐	☐	☐	☐	☐
Glossary Of Terms Definition	1		Task Descriptor	☐	☐	☐	☐	☐	☐
Business Logic Design	2		Task Descriptor	☐	☐	☐	☐	☐	☐
Data Persistence Design	3	2	Activity	✓	☐	☐	☐	☐	☐
Database Model Specification	4		Task Descriptor	☐	☐	☐	☐	☐	☐
Database Scripts Generation	5	4	Task Descriptor	☐	☐	☐	☐	☐	☐
Database Scripts Revision	6	5	Task Descriptor	☐	☐	☐	☐	☐	☐

FIGURE 5.5: The process editor of MOSKitt4ME

which is called "Business Logic Design", has an output product that is called "UML Class Model".

Unlike method content elements, processes are defined by means of a process editor, which is also opened by selecting the "Edit" action (or double-clicking the process elements). By means of this editor, processes are defined as work breakdown structures. These work breakdown structures define activities, which contain references to method content elements. Activities can be created by means of the contextual menu; references to content elements can be created by adding task descriptors, role descriptors, and work product descriptors. Task descriptors can be added by means of the contextual menu; work product descriptors and role descriptors are automatically created according to the roles and work products that are associated to the tasks in the content packages. Finally, precedence relationships can be established between process elements by means of the "Predecessors" column. Figure 5.5 shows an example of a process that has been defined by means of the process editor of the EPF Composer.

The SPEM2BPMN Transformation

This transformation allows method engineers to automatically obtain BPMN 2.0 representations of the method processes. To invoke this transformation, the user must open the contextual menu of the root element and select "Open BPMN 2.0 Diagram". This action generates a BPMN 2.0 process for each SPEM 2.0 activity.

The Activiti Designer is the editor that has been integrated in MOSKitt4ME to support BPMN 2.0. By means of this editor, method engineers can manually

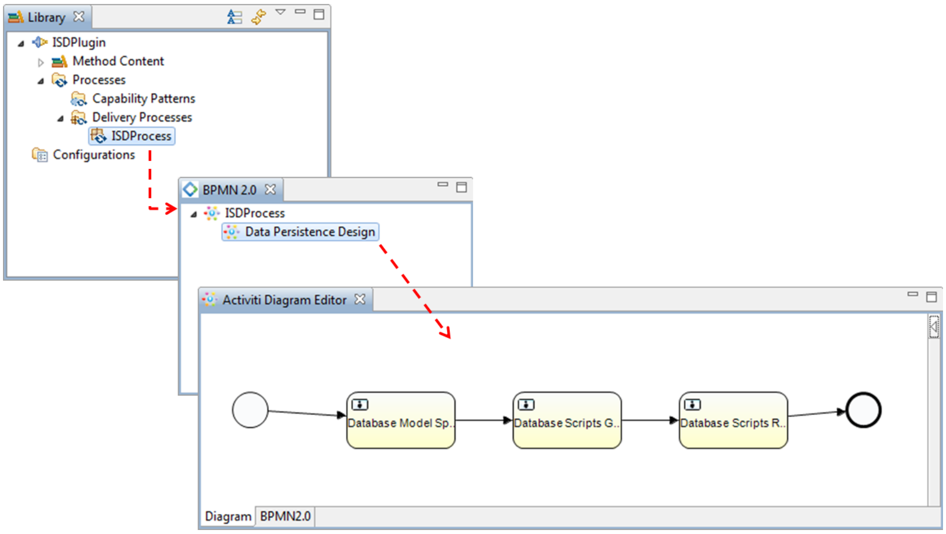


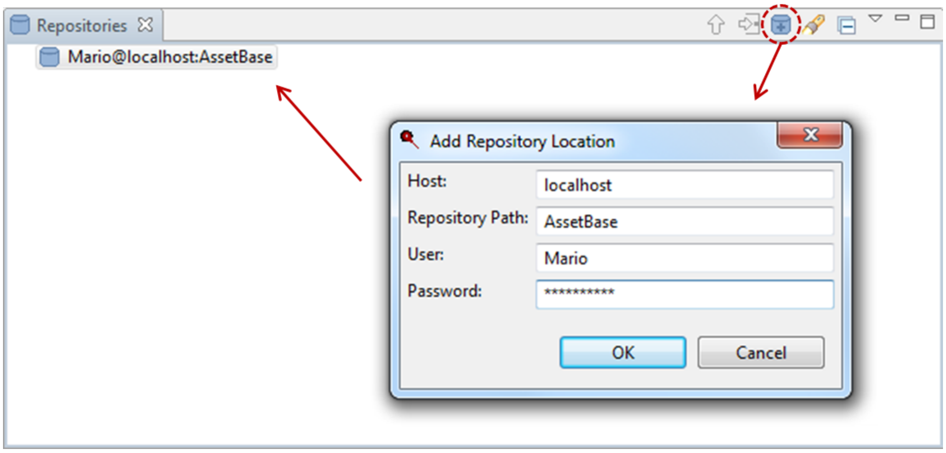
FIGURE 5.6: The BPMN 2.0 view

modify the generated processes; for instance, they can add BPMN 2.0 gateways to enhance the process workflows.

In order to facilitate the access to the BPMN 2.0 processes, MOSKitt4ME provides the “BPMN 2.0” view. This view can be opened by means of the “Show View” dialog, which is available at the “Window” menu. The BPMN 2.0 view provides a hierarchical representation of the BPMN 2.0 processes that are generated for the process that is selected in the Library view. Each of the elements of the hierarchy represents one specific BPMN 2.0 process. The user can double-click any of these processes to open the Activiti Designer file storing it. This is illustrated in Figure 5.6.

5.3.1.2 Connecting to Repositories: the Repository Client

The repository client of MOSKitt4ME can be accessed by opening the “Repositories” view. This view can be opened by means of the “Show View” dialog, which is available at the “Window” menu. The Repositories view allows method engineers to connect to FTP repositories that store either conceptual fragments or technical fragments. Thus, the main functionality that is provided by the Repositories view



Host: localhost
Repository Path: AssetBase
User: Mario
Password: **********

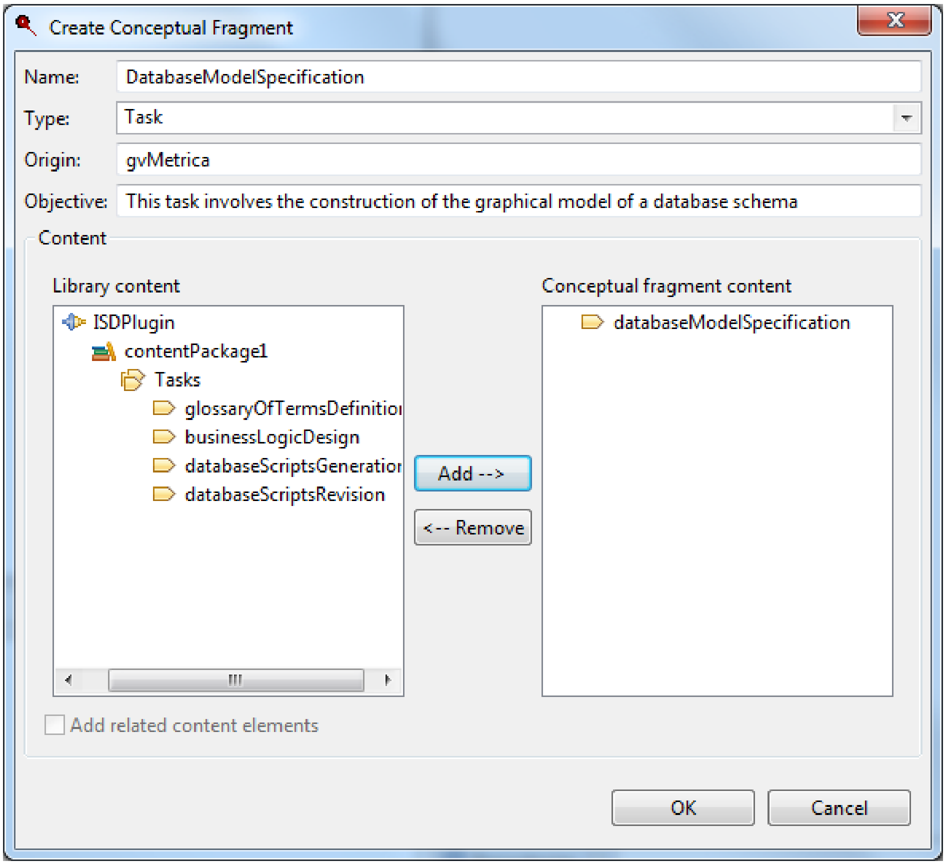
FIGURE 5.7: Adding a repository location in MOSKitt4ME

is twofold. On the one hand, it allows method engineers to reuse method parts, which makes the method construction process less error-prone, and more rapid and cost-effective. On the other hand, it allows method engineers to establish the linkage between the conceptual aspects and the technical aspects of methods; that is, it enables the association of conceptual method elements with the technical fragments that indicate the software tools that will support the method during its execution.

In order to connect to a FTP repository, the Repositories view provides the “Add Repository Location” action, which can be found in the toolbar of the view. This action opens the “Add Repository Location” dialog, which allows the user to enter the host that contains the repository, the repository path, and his/her username and password. The “Add Repository Location” dialog, together with the Repositories view, is shown in Figure 5.7. As the figure shows, repository locations are displayed in the Repositories view according to the following pattern: *user* @ *host* : *repositorypath*. These locations can be refreshed or deleted by means of the contextual menu.

Conceptual Fragments

The Repositories view allows method engineers to store new conceptual fragments in the FTP repositories by means of the “Create Conceptual Fragment” action of the contextual menu. This action opens the “Create Conceptual Fragment”


Create Conceptual Fragment

Name: DatabaseModelSpecification

Type: Task

Origin: gvMetrica

Objective: This task involves the construction of the graphical model of a database schema

Content

Library content

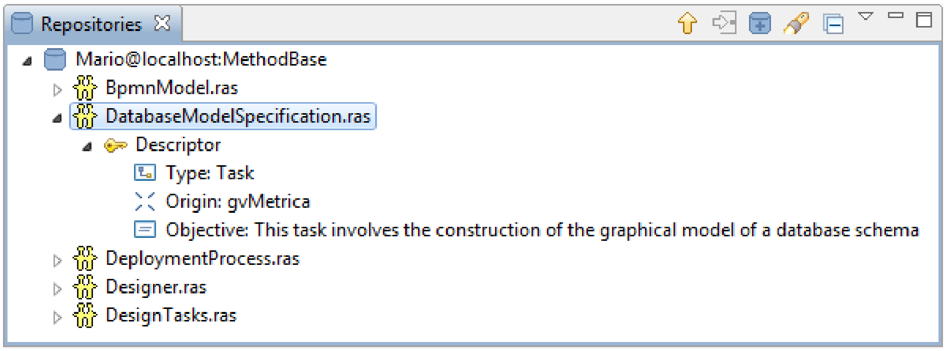
- ISDPlugin
  - contentPackage1
    - Tasks
      - glossaryOfTermsDefinitio
      - businessLogicDesign
      - databaseScriptsGeneratio
      - databaseScriptsRevision

Conceptual fragment content

- databaseModelSpecification

FIGURE 5.8: Dialog for creating conceptual fragments

dialog, which is graphically depicted in Figure 5.8. As the figure shows, this dialog allows the user to specify the fragment properties (e.g., name, type, and origin) and also to select the elements of the method that will be encapsulated in the conceptual fragment. The elements of the method can be selected by means of a graphical component that is called “Content”. This component filters the elements according to the fragment type; for instance, in Figure 5.8 the graphical component only displays the method tasks because the user is going to create a task fragment (i.e., the user has selected “Task” as the fragment type). We have also implemented the possibility to store different kinds of content elements (e.g., two products and one task) in the same conceptual fragment. To create this type of conceptual fragment, the user must select “Content Element” as the fragment



Mario@localhost:MethodBase
BpmnModel.ras
DatabaseModelSpecification.ras
Descriptor
Type: Task
Origin: gvMetrica
Objective: This task involves the construction of the graphical model of a database schema
DeploymentProcess.ras
Designer.ras
DesignTasks.ras

FIGURE 5.9: Conceptual fragment properties in the Repositories view

type.

When a conceptual fragment is created and stored in the repository, the fragment is shown as a nested element of the repository location. This is illustrated in Figure 5.9. As the figure shows, the Repositories view also allows method engineers to examine the fragment properties by opening the nested elements of the conceptual fragment. These properties can be used by method engineers to search for the most appropriate fragments. To facilitate this search, the Repositories view provides the "Search" action, which can be found in the toolbar of the view. This action opens a dialog that allows method engineers to specify different property values, which are used by the Repositories view to filter its content (i.e., the Repositories view will only show the fragments whose properties match the values that are specified by the user).

Once the user selects a conceptual fragment in the Repositories view, the integration of the fragment into the method under construction can be performed by means of the "Integrate Conceptual Fragment" action, which can be found in the toolbar of the view. The integration of a conceptual fragment is performed in two different ways depending on the type of the fragment that is being integrated. If the fragment is a content fragment (i.e., a task fragment, a role fragment, a product fragment, or contains a combination of these), then the "Integrate Conceptual Fragment" action will open the "Content Package Selection" dialog. This dialog allows the user to select the content package that will store the content elements that are encapsulated in the fragment. On the other hand, if the fragment that is

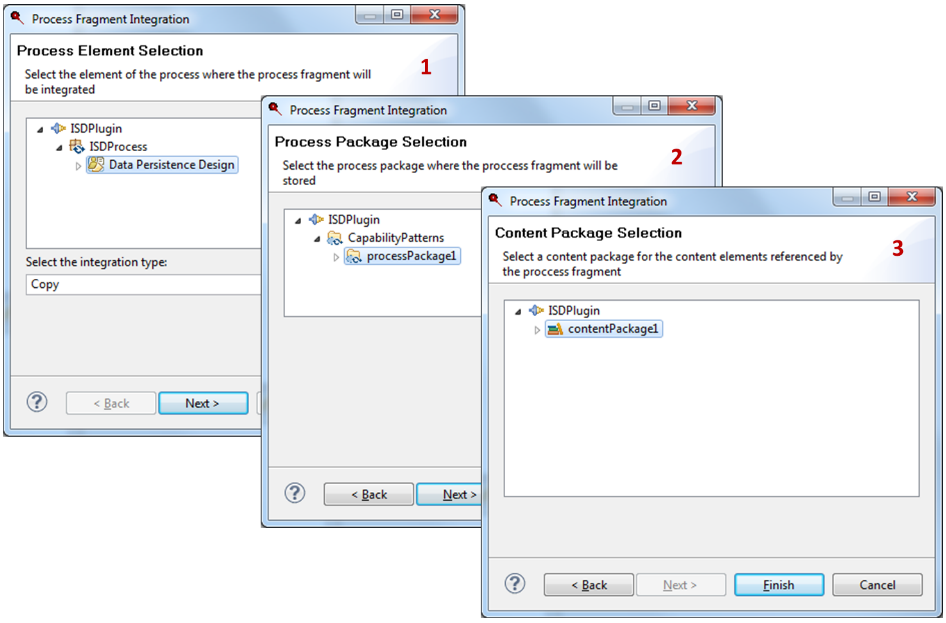

Process Fragment Integration
Process Element Selection
Select the element of the process where the process fragment will be integrated
1
ISDPlugin
ISDProcess
Data Persistence Design
Select the integration type:
Copy
< Back
Next >
Process Fragment Integration
Process Package Selection
Select the process package where the proccess fragment will be stored
2
ISDPlugin
CapabilityPatterns
processPackage1
< Back
Next >
Process Fragment Integration
Content Package Selection
Select a content package for the content elements referenced by the proccess fragment
3
ISDPlugin
contentPackage1
< Back
Next >
Finish
Cancel

FIGURE 5.10: Wizard for process fragment integration

selected is a process fragment, then the "Integrate Conceptual Fragment" action will open the "Process Fragment Integration" wizard.

The "Process Fragment Integration" wizard is depicted in Figure 5.10. As the figure shows, the wizard takes the user through three sequential steps. In the first step, the user must select an element of a method process. This element will be the destination of the process that is encapsulated in the fragment (e.g., if the selected element is an activity, then the process will be nested within this activity). The integration of the process fragment can be performed in two different ways: "Extend" and "Copy". If "Extend" is selected, then the process fragment will not be included in the method process but rather copied separately and referenced; if "Copy" is selected, then all of the fragment content will be copied into the selected process element. Once the first step of the wizard is finished, in the second step the user must select the process package where the process that is encapsulated in the fragment will be stored. If the process already exists in the package, the process will not be copied. In this case, the third step of the wizard is omitted; otherwise, the third step involves the selection of a content package of the method

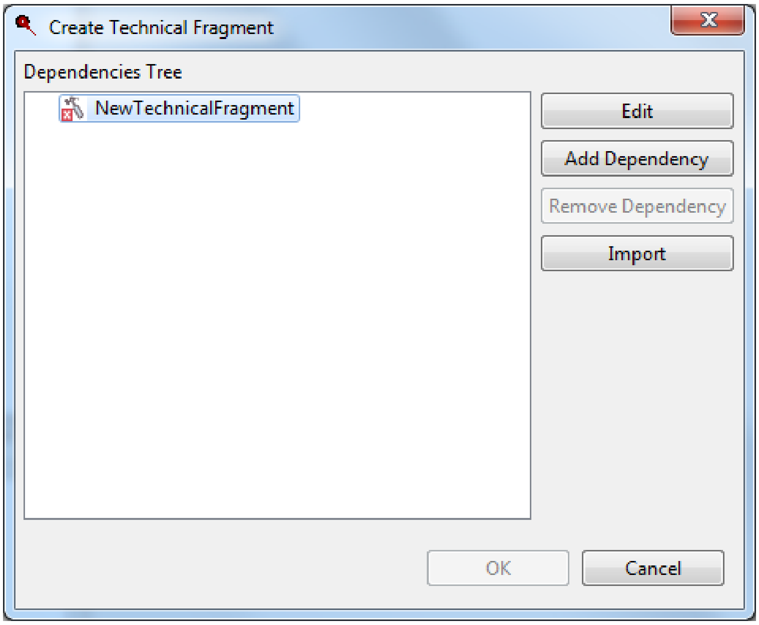
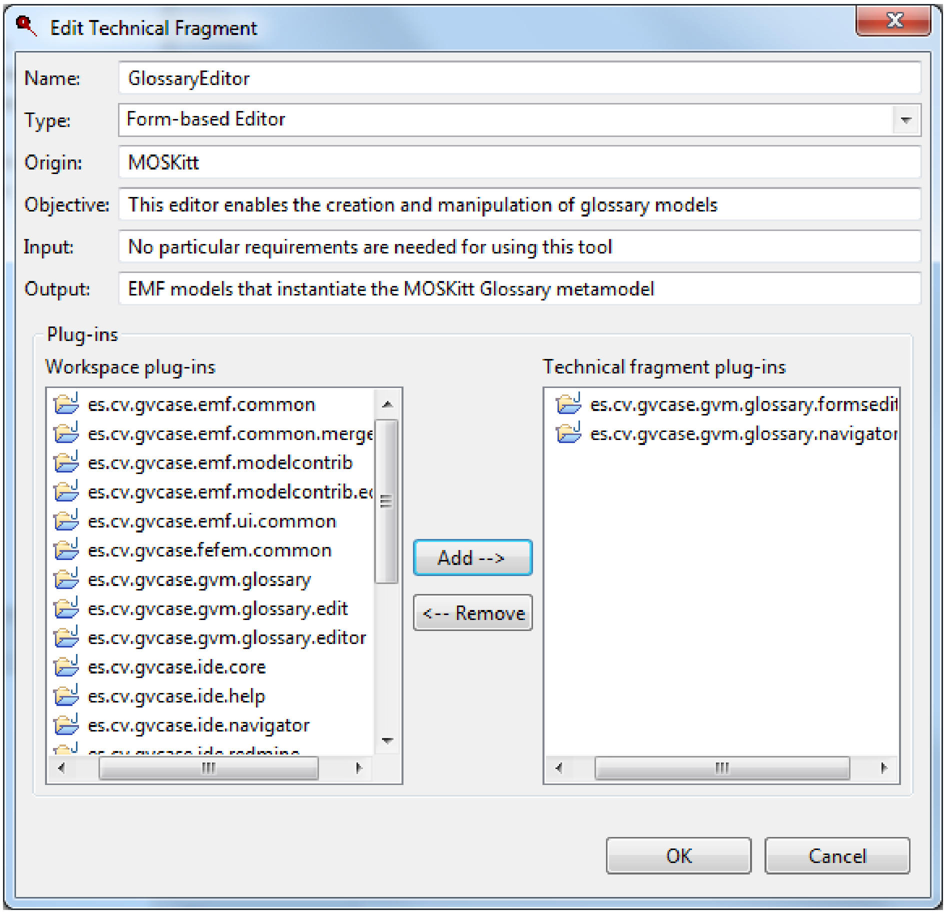


FIGURE 5.11: Dialog for technical fragment creation

library. This package will store the content elements that are referenced by the process fragment.

Technical Fragments

In addition to conceptual fragments, the Repositories view also enables the creation of technical fragments. To do this, method engineers must invoke the "Create Technical Fragment" action, which is available in the contextual menu of the repository location. This action opens the "Create Technical Fragment" dialog, which is graphically depicted in Figure 5.11. As the figure shows, this dialog allows the user to create new technical fragments and also to specify their dependencies with other technical fragments; that is, the dependency tree (see Section 4.2.2.1). Specifically, the "Create Technical Fragment" dialog supports the creation of technical fragments by means of a graphical component that is called "Dependencies Tree". This component, which implements a Tree Viewer of the Eclipse JFace library, initially contains one technical fragment (i.e., the root of the dependency tree) and also allows the user to add additional fragments representing its software dependencies. This is done by means of the "Add Dependency"


Edit Technical Fragment

Name: GlossaryEditor

Type: Form-based Editor

Origin: MOSKitt

Objective: This editor enables the creation and manipulation of glossary models

Input: No particular requirements are needed for using this tool

Output: EMF models that instantiate the MOSKitt Glossary metamodel

Plug-ins

Workspace plug-ins

- es.cv.gvcase.emf.common
- es.cv.gvcase.emf.common.merge
- es.cv.gvcase.emf.modelcontrib
- es.cv.gvcase.emf.modelcontrib.ec
- es.cv.gvcase.emf.ui.common
- es.cv.gvcase.fefem.common
- es.cv.gvcase.gvm.glossary
- es.cv.gvcase.gvm.glossary.edit
- es.cv.gvcase.gvm.glossary.editor
- es.cv.gvcase.ide.core
- es.cv.gvcase.ide.help
- es.cv.gvcase.ide.navigator

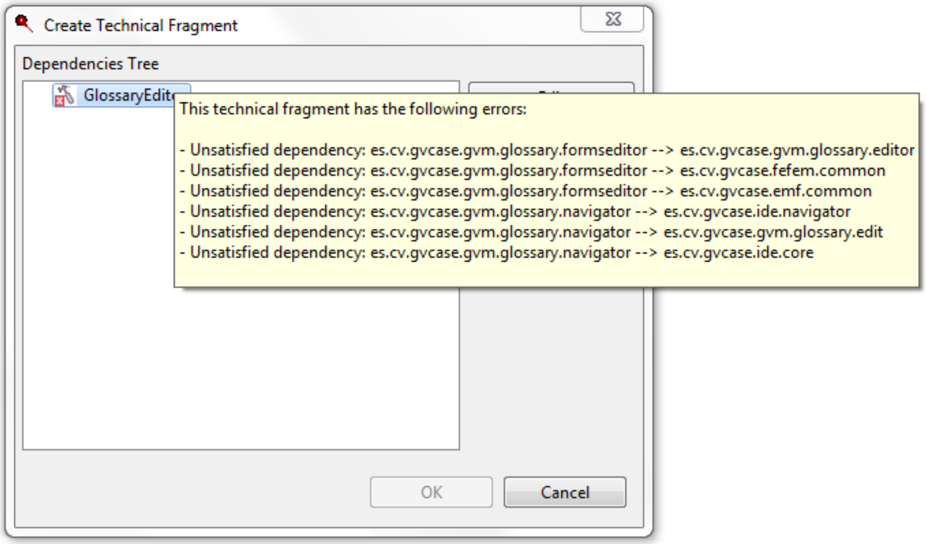
Technical fragment plug-ins

- es.cv.gvcase.gvm.glossary.formsedit
- es.cv.gvcase.gvm.glossary.navigator

FIGURE 5.12: Dialog for the edition of technical fragments

button. On the other hand, the “Remove Dependency” button allows the user to remove technical fragments from the tree.

In order to edit the properties of the technical fragment that is selected in the dependency tree, the “Create Technical Fragment” dialog provides the “Edit” button, which opens the “Edit Technical Fragment” dialog. This dialog, which is graphically depicted in Figure 5.12, allows the user to specify the properties of the technical fragment (e.g., name, type, and input/output) and also to select the Eclipse plug-ins that implement the tool that will be encapsulated in the fragment. These plug-ins can be selected by means of a graphical component that is called



This technical fragment has the following errors:

- Unsatisfied dependency: es.cv.gvcase.gvm.glossary.formseditor --> es.cv.gvcase.gvm.glossary.editor
- Unsatisfied dependency: es.cv.gvcase.gvm.glossary.formseditor --> es.cv.gvcase.fefem.common
- Unsatisfied dependency: es.cv.gvcase.gvm.glossary.formseditor --> es.cv.gvcase.emf.common
- Unsatisfied dependency: es.cv.gvcase.gvm.glossary.navigator --> es.cv.gvcase.ide.navigator
- Unsatisfied dependency: es.cv.gvcase.gvm.glossary.navigator --> es.cv.gvcase.gvm.glossary.edit
- Unsatisfied dependency: es.cv.gvcase.gvm.glossary.navigator --> es.cv.gvcase.ide.core

FIGURE 5.13: Technical fragments errors

“Plug-ins”. This component displays all of the projects of type “Plug-in Project” that are available in the MOSKitt4ME workspace.

Once the fragment properties are specified, errors may appear due to, for example, software dependencies (i.e., the plug-ins of the technical fragment may require other plug-ins that are not included in the fragment); another example of error is when the user does not specify the name or the type of the technical fragment. A technical fragment that contain errors is highlighted in the dependency tree by means of a red “X”. This is illustrated in Figure 5.13. In order to see all of the errors of a technical fragment, the user can place the cursor over the fragment area, as illustrated in the figure.

In order to solve the errors that are related to software dependencies, the user can add/remove technical fragments in the dependency tree by means of the “Add Dependency” and “Remove Dependency” buttons. When the user pushes the “Add Dependency” button, a new technical fragment is created. This fragment is added to the tree as a child of the fragment that is selected by the user; thus, the child fragment represents a dependency of the parent fragment (i.e., the parent fragment requires the use of the child fragment for its correct operation). On the other hand, the “Remove Dependency” button allows the user to remove the

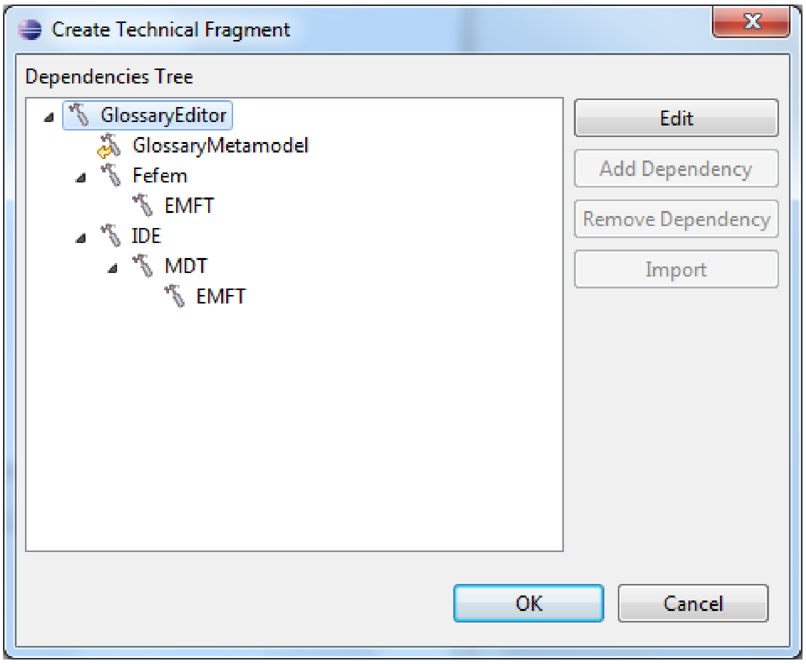
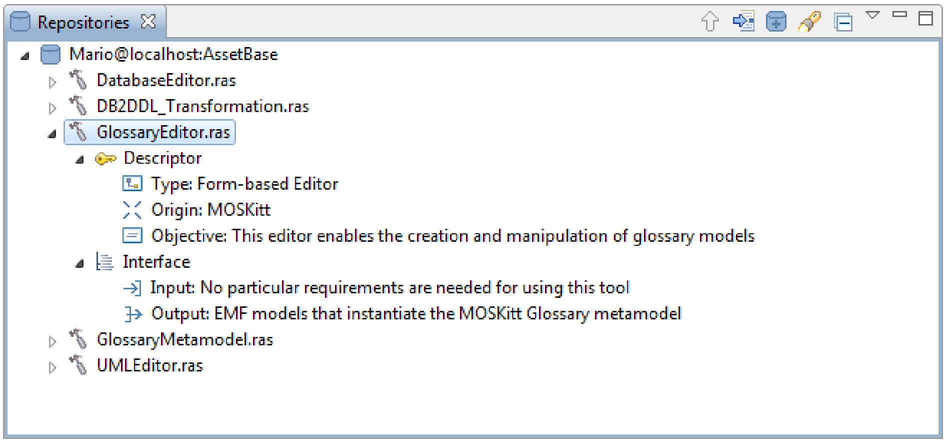


FIGURE 5.14: An example of dependency tree

fragment that is selected in the dependency tree and also all of its nested fragments (i.e., all of its software dependencies).

In addition to the add/remove buttons, the user can establish dependencies between technical fragments by means of the "Import" button. This button opens the "Import Technical Fragment" dialog, which allows the user to select a technical fragment that is already stored in the repository and to import it (and all of its software dependencies) into the dependency tree. When a fragment is imported, it is added to the dependencies tree as a child of the selected fragment. Unlike regular fragments, the icon of the imported fragments contains a yellow arrow. This arrow represents that these fragments are references to fragments that already exist in the repository, and, therefore, they must not be created so as to avoid duplicate fragments. As an example, Figure 5.14 shows the dependency tree of the "Glossary Editor" fragment. Note that, in the example, the "Glossary Metamodel" is the only fragment that has been imported from the repository. Additionally, none of the technical fragments contain errors, and, therefore, the



Mario@localhost:AssetBase
DatabaseEditor.ras
DB2DDL_Transformation.ras
GlossaryEditor.ras
Descriptor
Type: Form-based Editor
Origin: MOSKitt
Objective: This editor enables the creation and manipulation of glossary models
Interface
Input: No particular requirements are needed for using this tool
Output: EMF models that instantiate the MOSKitt Glossary metamodel
GlossaryMetamodel.ras
UMLEditor.ras

FIGURE 5.15: Technical fragment properties in the Repositories view

user can push the "OK" button to create a new technical fragment for each of the fragments of the tree (with the exception of the imported fragment).

Similarly to conceptual fragments, when a technical fragment is created and stored in the repository, the fragment is shown as a nested element of the repository location. This is illustrated in Figure 5.15. As the figure shows, the Repositories view also allows method engineers to examine the fragment properties by opening the nested elements of the technical fragment. These properties can be used by method engineers to search for the most appropriate fragments. This task is facilitated by the "Search" action of the toolbar.

Note that, some technical fragments are not shown in the Repositories view. This is because technical fragments may be created for the sole purpose of resolving dependency problems, and, thus, they may fall out of the tool types that are supported by MOSKitt4ME (e.g., editors, meta-models, model transformations, or guidance). To create this type of fragments, MOSKitt4ME provides the category "Others".

Once the user selects a technical fragment in the Repositories view, the linkage between this fragment and a conceptual element of the method can be established by means of the "Integrate Technical Fragment" action. This action, which can be found in the toolbar of the view, opens the "Task/Work Product Selection" dialog. This dialog, which is depicted in Figure 5.16, allows the user to select a task or

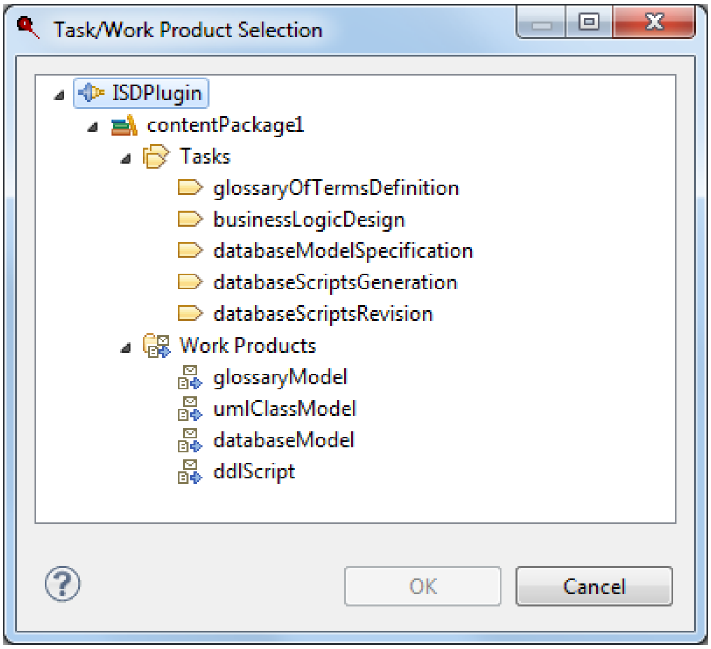
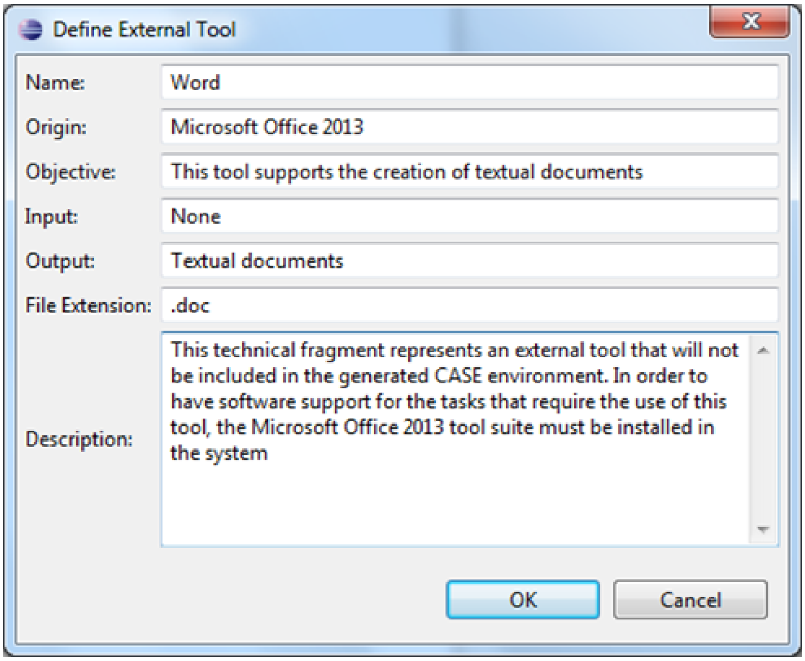


FIGURE 5.16: Dialog for establishing conceptual-technical linkage

a product of the method. When a task or a product is chosen, it is associated to the technical fragment that is selected in the Repositories view. This association is performed by means of the creation of a new element of type "Tool Mentor", which is automatically displayed on the Library view.

Finally, it is also important to mention that the Repositories view offers two additional actions in the contextual menu of the repository location: "Define External Tool" and "Define Internal Tool". These actions allow the user to create two special types of technical fragments: external tools and internal tools. External tools differ from regular technical fragments in that they do not contain Eclipse plug-ins; rather, they represent software tools that are implemented in a different technology. Thus, external tools support situations where method engineers want to specify that a particular product must be developed by means of a tool that cannot be installed in an Eclipse-based environment; for instance, a product that needs to be created during the method execution by means of Microsoft Word. On the other hand, internal tools differ from regular technical fragments in that they

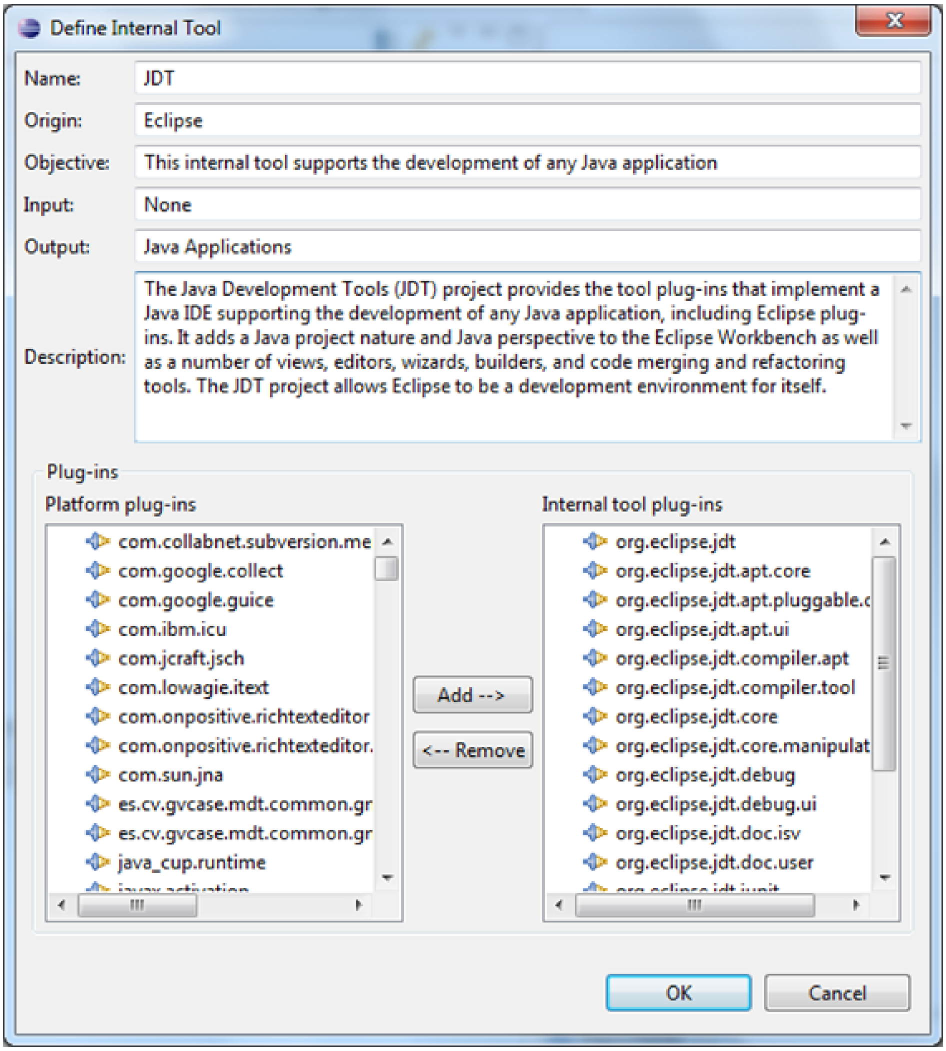


Name:	Word
Origin:	Microsoft Office 2013
Objective:	This tool supports the creation of textual documents
Input:	None
Output:	Textual documents
File Extension:	.doc
Description:	This technical fragment represents an external tool that will not be included in the generated CASE environment. In order to have software support for the tasks that require the use of this tool, the Microsoft Office 2013 tool suite must be installed in the system

FIGURE 5.17: Dialog for the definition of external tools

do not contain plug-ins that have been implemented by the user (which are developed by means of the metatools of MOSKitt4ME and located in the workspace); rather, they reference plug-ins that are installed in the MOSKitt4ME platform. Thus, internal tools allow method engineers to specify that a particular product will be created by means of an Eclipse-based framework such as GMF, EMF, or FEFEM. One important advantage of internal tools is that, since these tools are available in the MOSKitt4ME platform, it is not necessary to store their plug-ins in the repository; it suffices to store their identifiers. Another advantage of internal tools is that they are not subject to any requirements, and, therefore, they can represent any type of software tool that can be conceived as Eclipse plug-ins; nonetheless, this lack of requirements prevents the Project Manager of the CASE part of MOSKitt4ME to interact with the internal tools during the method execution.

For the definition of external tools, MOSKitt4ME opens the "Define External Tool" dialog after the user selects the "Define External Tool" action in the contextual menu. This dialog is shown in Figure 5.17. Similarly to the definition



Name: JDT

Origin: Eclipse

Objective: This internal tool supports the development of any Java application

Input: None

Output: Java Applications

Description: The Java Development Tools (JDT) project provides the tool plug-ins that implement a Java IDE supporting the development of any Java application, including Eclipse plug-ins. It adds a Java project nature and Java perspective to the Eclipse Workbench as well as a number of views, editors, wizards, builders, and code merging and refactoring tools. The JDT project allows Eclipse to be a development environment for itself.

Plug-ins

Platform plug-ins	Internal tool plug-ins
com.collabnet.subversion.me	org.eclipse.jdt
com.google.collect	org.eclipse.jdt.apt.core
com.google.guice	org.eclipse.jdt.apt.pluggable.c
com.ibm.icu	org.eclipse.jdt.apt.ui
com.jcraft.jsch	org.eclipse.jdt.compiler.apt
com.lowagie.itext	org.eclipse.jdt.compiler.tool
com.onpositive.richtexteditor	org.eclipse.jdt.core
com.onpositive.richtexteditor.	org.eclipse.jdt.core.manipulat
com.sun.jna	org.eclipse.jdt.debug
es.cv.gvcase.mdt.common.gr	org.eclipse.jdt.debug.ui
es.cv.gvcase.mdt.common.gr	org.eclipse.jdt.doc.isv
java_cup.runtime	org.eclipse.jdt.doc.user

Figure 5.18: Dialog for the definition of internal tools

of regular technical fragments, the “Define External Tool” dialog allows the user to specify the properties of the external tool (e.g., name, origin, and objective); nonetheless, to define an external tool, the user must also specify a file extension and a description. The former property represents the extension of the files that will be created by means of the external tool. The main purpose of this property is to be used by the operating system as a way to identify the software tool that must be launched during the method execution. In the example of Figure 5.17, the “.doc” extension identifies Microsoft Word as the tool that must be invoked for the creation of the products that are associated to the external tool. On the other hand, the latter property provides textual information that can be used by software engineers to configure the system so that the external tool is available during the method execution. This information is made available to the software engineer by means of a report that is produced during the CASE environment generation process.

The definition of internal tools can be performed in MOSKitt4ME by means of the “Define Internal Tool” dialog. This dialog, which is shown in Figure 5.18, allows the user to specify the properties of the internal tool, such as its name, origin, and description. In this case, the description property will provide textual information that can be used by software engineers during the method execution; note that, the Project Manager component of the CASE environment cannot provide any other assistance regarding internal tools because they do not implement any common interface. In addition to the above properties, method engineers must also specify the plug-ins to be referenced by the internal tool. To specify these plug-ins, the “Define Internal Tool” dialog provides a graphical component that is called “Plug-ins”. This component enables the selection of plug-ins from the whole MOSKitt4ME platform.

5.3.1.3 Building Eclipse Plug-ins: the Metatools

MOSKitt4ME provides method engineers with a set of Eclipse technologies that facilitate the construction of the plug-ins to be encapsulated in technical fragments. Since the major goal of these plug-ins is to provide product support during the method execution, it is desirable that they can communicate with the Project Manager of MOSKitt4ME. To enable this communication, the Eclipse plug-ins that are developed using the metatools of MOSKitt4ME must meet a set of requirements. In the following, we describe for all of the types of technical fragments that are supported in the current version of MOSKitt4ME, the requirements that they must satisfy and also the Eclipse technologies that can be used for their development. Note that we do not include external tools and internal tools in this section. This is because, unlike the other types of technical fragments, external/internal tools are not developed by MOSKitt4ME users.

Metamodels

Metamodels can be specified in MOSKitt4ME by means of the Ecore language, which is provided as part of the EMF framework. Thus, the EMF framework supports the definition of metamodels, and, additionally, it provides generation facilities that take metamodels as input and automatically obtain (1) the set of Java classes that implement the metamodels, along with (2) Java classes that enable the edition of the models (that instantiate the metamodels), and (3) basic tree-based editors.

In order to be compatible with the MOSKitt4ME requirements, the plug-ins that are contained in a technical fragment of type "Metamodel" must define an Ecore metamodel; additionally, the Java classes that implement the metamodel, the editing classes, and the tree-based editor must have been generated.

Graphical Editors

Graphical editors can be developed in MOSKitt4ME by means of the GMF framework. This framework, which is built on EMF, applies a model-driven approach to obtain fully-functional graphical editors. To develop a graphical editor using GMF, the user must build a set of models that define (1) an Ecore metamodel, (2) the graphical elements to be displayed in the editor, and (3) the tools that will appear in the palette, menus, and toolbars. Once these models are defined, a set of generative tools automatically obtain a graphical editor supporting the construction of models that are compliant with the metamodel that has been specified.

There are no particular requirements to be met by the plug-ins that are contained in a technical fragment of type "Graphical Editor". All of the graphical editors that are developed by means of GMF are compatible with the Project Manager of MOSKitt4ME.

Form-based Editors

Form-based editors can be developed in MOSKitt4ME by means of the MOSKitt FEFEM framework. This framework facilitates the construction of form-based editors by implementing a set of patterns that are typically found during the development of this kind of editors. An example of these patterns is a textbox for editing properties of type String. FEFEM offers a class that represents this pattern; developers only need to extend this class and write a few lines of code to add this kind of textboxes to their editors. It is important to note that, since FEFEM is based on SWT and JFace, form-based editors can also be directly built by means of these libraries; however, we recommend the use of FEFEM since it significantly reduces the workload that is inherent to the development of form-based editors.

There are no particular requirements to be met by the plug-ins that are contained in a technical fragment of type "Form-based Editor". All of the form-based editors that are developed by means of FEFEM (or directly by means of SWT and JFace) are compatible with the Project Manager of MOSKitt4ME.

Model Transformations

For the development of model transformations, MOSKitt4ME offers the ATL and Xpand languages, which are two programming languages that have been designed by the Eclipse community to support the implementation of M2M and M2T transformations, respectively.

Despite the availability of these two languages in MOSKitt4ME, model transformations can be implemented in any language (for instance, Java or C++); nonetheless, to be compatible with the Project Manager component, the plug-ins that are contained in a technical fragment of type "Model Transformation" must meet two requirements:

- The model transformation must be declared by means of the extension point "es.cv.gvcase.trmanager.transformation", which is provided by the Transformation Manager of MOSKitt. The Transformation Manager is a software component that provides a set of Java classes that facilitate the specification and the invocation of model transformations; additionally, it implements a graphical user interface that makes these transformations readily available in the MOSKitt workbench.

- In the declaration of the model transformation by means of the aforementioned extension point, the user must provide a Java class extending the "Transformation" class of the "es.cv.gvcase.trmanager" plug-in. This class is an abstract class that declares two abstract methods: "transform" and "inputsValid". While the former must implement the invocation of the model transformation, the latter must implement validation rules for the input model. Thus, by extending the "Transformation" class, the Project Manager will be able to invoke the model transformation during the method execution by simply calling the "transform" method.

Guidance

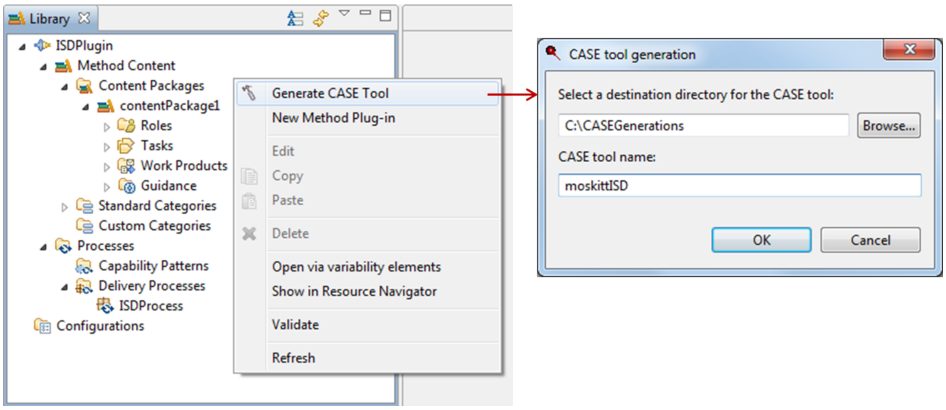
Contextual help can be developed in MOSKitt4ME by means of the HTML and XML languages. This help can be later encapsulated in guidance fragments. In order for these guidance fragments to be compatible with MOSKitt4ME, the Eclipse plug-ins that implement the help must use two extension points:

- *org.eclipse.epf.authoring.ui.helpcontextprovider:* the plug-ins of a guidance fragment must use this extension point to declare an identifier for the contextual help. During method execution, whenever a method task that is associated to the guidance fragment is selected in the Process view, this identifier will be sent to the Help view of the Project Manager. Thus, the Help view knows the help that must be shown to the user based on the task that he/she is going to execute.

- *org.eclipse.help.contexts:* the plug-ins of a guidance fragment must use this extension point to declare a "contexts.xml" file. This file associates the help identifier (which is defined by means of the previous extension point) with the HTML/XML files that implement the help.

5.3.1.4 Obtaining Software Support: the CASE Generator

The CASE generator of MOSKitt4ME can be accessed by means of the "Generate CASE Tool" action of the contextual menu of the Library view. This action, which



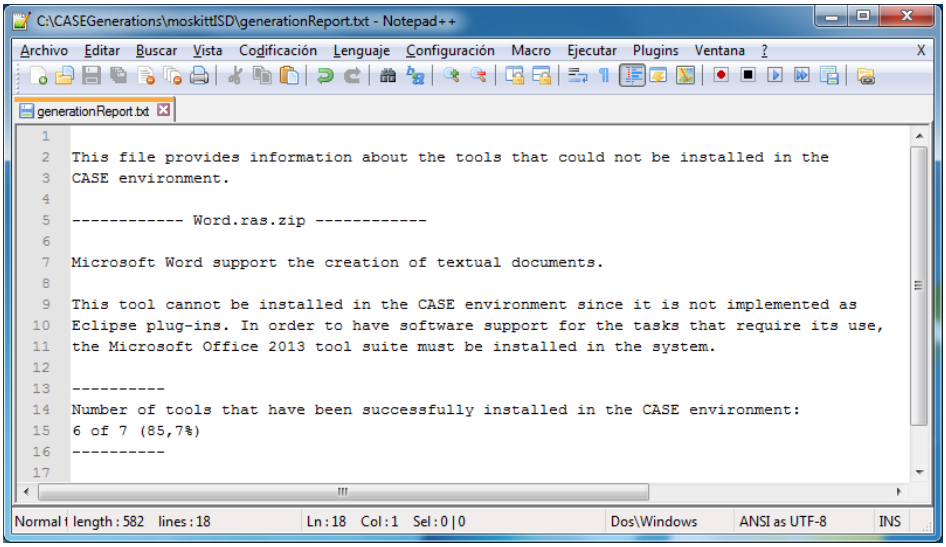
Select a destination directory for the CASE tool: C:\CASEGenerations

CASE tool name: moskittISD

FIGURE 5.19: Generation of CASE environments in MOSKitt4ME

is shown on the left side of Figure 5.19, opens the "CASE Tool Generation" dialog, which is shown on the right side. The "CASE Tool Generation" dialog allows the user to enter the destination path and the name of the software environment that will be generated. When the user pushes the "OK" button, the generation process starts. Nonetheless, the executable representation of the method must have been previously obtained; otherwise, the dialog will show an error message suggesting that the user must invoke the "Open BPMN 2.0 Diagram" action.

When the user successfully starts the generation process, three automatic steps are performed. First, the technical fragments that are associated to method elements are downloaded from the FTP repository and stored in a temporal local folder. In this step, all of the dependencies of these fragments are also downloaded to the same folder. This first step corresponds to the first two steps that are described in Section 4.3.1. Second, the Eclipse plug-ins that are contained in the technical fragments are extracted from the *.ras* zip files and imported into the MOSKitt4ME workspace (if they do not already exist). At this point, an extra plug-in is generated. This plug-in contains an Eclipse product configuration file that defines the RCP product (i.e., the CASE environment) that will be generated. Finally, in the third step of the process, an Eclipse product export process is launched. This process produces the final Eclipse-based environment, which is a reconfiguration of MOSKitt that only contains the plug-ins that are strictly necessary to support the method. These plug-ins are the ones that have been imported into the workspace along with the plug-ins that implement the Project


C:\CASEGenerations\moskittISD\generationReport.txt - Notepad++
Archivo Editar Buscar Vista Codificación Lenguaje Configuración Macro Ejecutar Plugins Ventana ?
generationReport.txt

```

This file provides information about the tools that could not be installed in the
CASE environment.

------------ Word.ras.zip ------------

Microsoft Word support the creation of textual documents.

This tool cannot be installed in the CASE environment since it is not implemented as
Eclipse plug-ins. In order to have software support for the tasks that require its use,
the Microsoft Office 2013 tool suite must be installed in the system.

----------
Number of tools that have been successfully installed in the CASE environment:
6 of 7 (85,7%)
----------

```

Normal t length : 582 lines : 18 Ln : 18 Col : 1 Sel : 0 | 0 Dos\Windows ANSI as UTF-8 INS

FIGURE 5.20: An example of generation report

Manager and a plug-in that contains the libraries of the Activiti Engine. Once the MOSKitt reconfiguration is completely generated, the plug-ins that were imported into the MOSKitt4ME workspace are deleted so that the workspace returns to its original state. This third step of the generation process corresponds to the last step that is described in Section 4.3.1.

The duration of the three steps that are described above depends on the size and the number of software tools that are required to support the method. Thus, the CASE environment generation process may take from a few minutes to more than one hour.

As a result of the generation process, a generation report is also produced (see Section 4.3.1.3). This report contains information about the number of tools that are successfully installed in the CASE environment; it also contains information about the external tools since these tools cannot be included in the environment, and, therefore, software engineers need to configure the system to make the external tools available during the method execution. Figure 5.20 shows an example of generation report.

5.3.2 The CASE Part of MOSKitt4ME

The software environments that are generated by means of the CASE generator (see Section 5.3.1.4) constitute the CASE part of the MOSKitt4ME architecture. This CASE part offers customized tool support that assists software engineers during the entire process of software development; that is, during the execution of the method that method engineers defined using the CAME part of MOSKitt4ME. All of the functionality of the CASE part is made available to software engineers through a component that is called Project Manager. This component is detailed in Section 5.3.2.1.

5.3.2.1 Bringing Methods into Enactment: the Project Manager

The project manager of MOSKitt4ME can be accessed by opening a perspective that is called "Method Execution". This perspective can be opened by means of the "Open Perspective" dialog, which is available at the "Window" menu. When the "Method Execution" perspective is opened, the MOSKitt4ME workbench is organized in four different parts, which are graphically depicted in Figure 5.21. These four parts are the following:

1. *MOSKitt Resource Explorer:* This view provides a hierarchical picture of the workspace. This hierarchy is organized in projects, folders, and files. From the MOSKitt Resource Explorer, software engineers can, for example, create new projects, delete existing projects, or add files to these projects. This view corresponds to the Project Explorer view that is described in Section 4.4.1.

2. *Process view:* This view shows the current state of the process instance that is associated to the project that is selected in the MOSKitt Resource Explorer. The process instance is shown in the same notation as the one that was used by method engineers; that is, the notation of the EPF Composer. The Process view corresponds to the view of the same name that is described in Section 4.4.1.

3. *Product Explorer:* This view shows a hierarchical picture of the artifacts that have been produced during the course of the project that is selected in

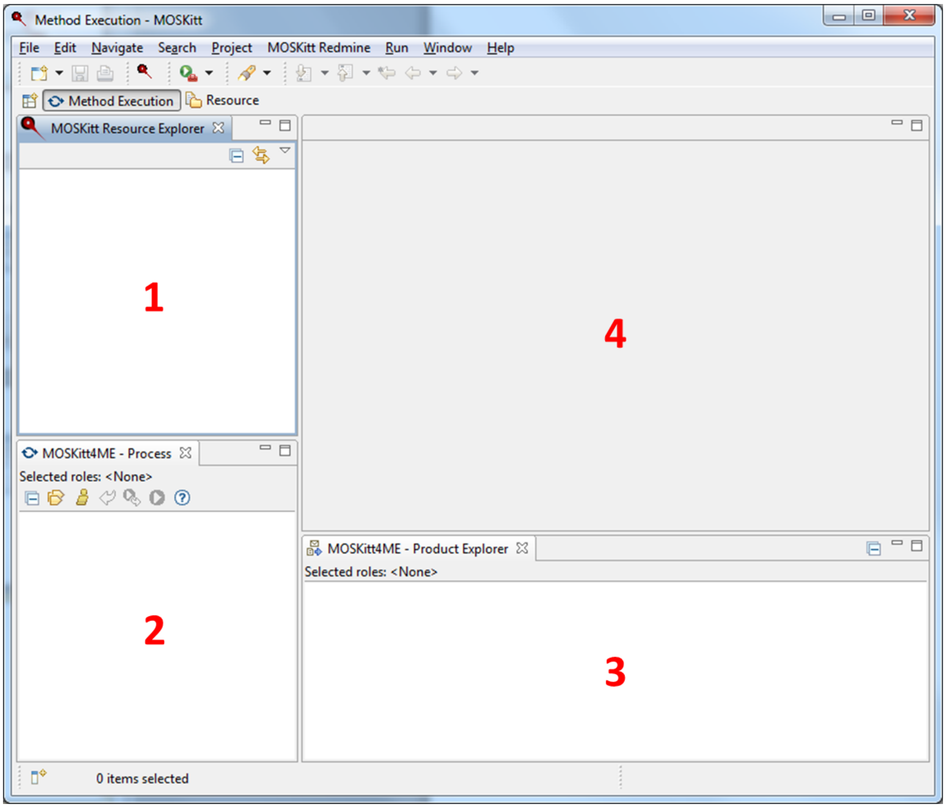
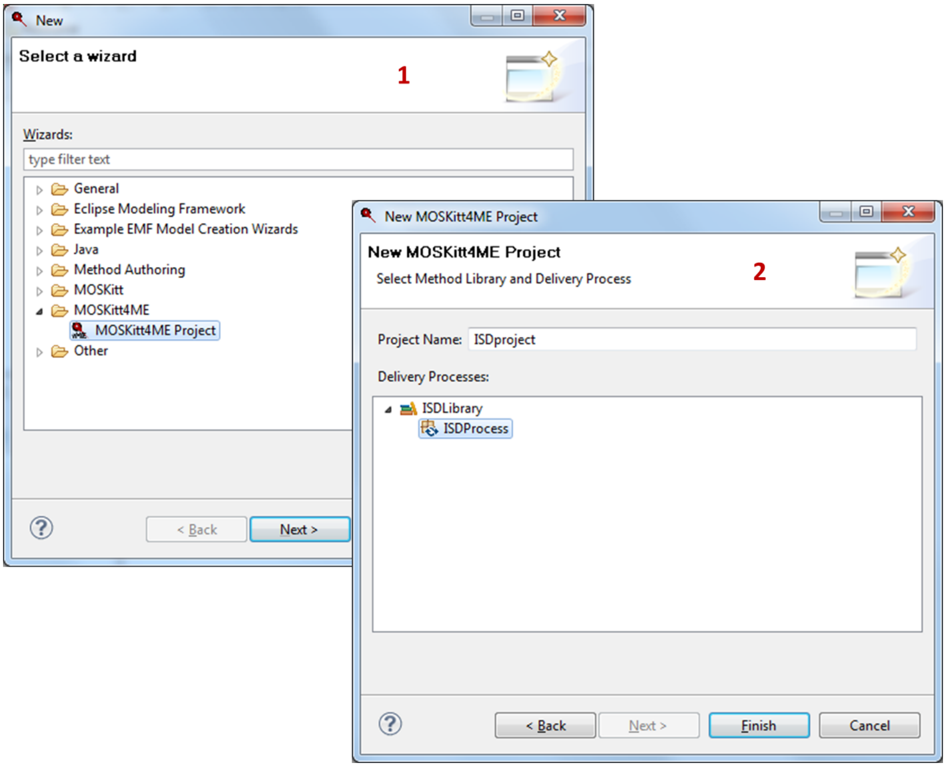

Method Execution - MOSKitt
File Edit Navigate Search Project MOSKitt Redmine Run Window Help
Method Execution
Resource
MOSKitt Resource Explorer
1
4
MOSKitt4ME - Process
Selected roles: <None>
2
MOSKitt4ME - Product Explorer
Selected roles: <None>
3
0 items selected

FIGURE 5.21: Parts of the method execution perspective

the MOSKitt Resource Explorer. This view corresponds to the view of the same name that is described in Section 4.4.1.

4. *Editor area:* Method products can be created and manipulated by means of software tools that are opened by default on the upper-right side of the MOSKitt4ME workbench. These tools correspond to the technical fragments (e.g., graphical or form-based editors), which were associated to the method tasks and products during the phase of method design.

In addition to the above views, which are opened by default when the "Method Execution" perspective is active, the project manager of MOSKitt4ME also provides two additional views. These views, which are in charge of showing method guidelines to software engineers, correspond to the Help view that is described in



Project Name: ISDproject

FIGURE 5.22: Wizard to create new projects in MOSKitt4ME

Section 4.4.1. Below, we detail all of the views that comprise the project manager of MOSKitt4ME.

MOSKitt Resource Explorer

The MOSKitt Resource Explorer allows software engineers to create and delete development projects. To create a new project, the user must select "Other..." in the "New" item of the contextual menu. This action opens the "New" wizard, which is graphically depicted in Figure 5.22. As the figure shows, the "New" wizard takes the user through two sequential steps. First, the user must select the project type; in this case, the type to select is called "MOSKitt4ME Project". Second, the user must enter the project name and select the process to be associated to the project. In this second page of the wizard, the user is allowed to choose among all of the processes that were defined during the phase of method design. When the wizard is finished, a new MOSKitt4ME project is added to the

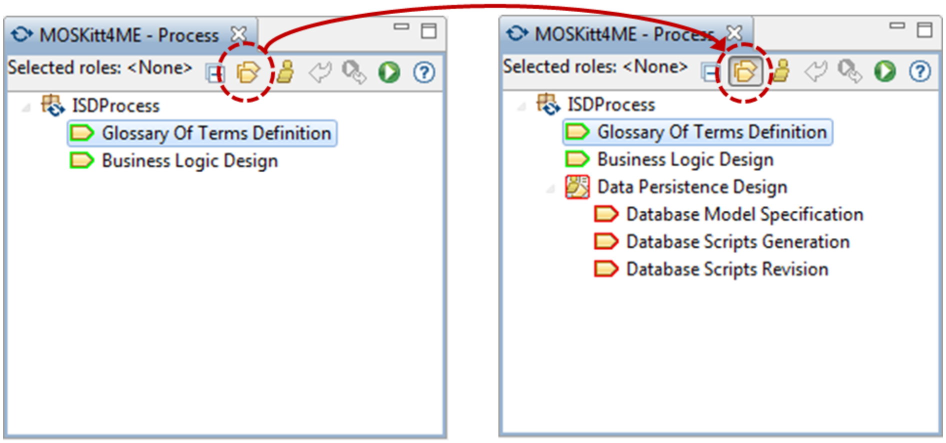


FIGURE 5.23: Task filtering in the process view

MOSKitt Resource Explorer view. This project can be deleted by means of the "Delete" action of the contextual menu.

Process View

When a project is selected in the MOSKitt Resource Explorer, the project manager of MOSKitt4ME automatically invokes the Activiti Engine, which returns the current state of the process instance that is associated to the project. The view that is in charge of showing this process instance is the Process view.

By default, the Process view only shows the tasks that are executable in the current state of the process. These tasks (and their parent activities) are displayed in green. This is illustrated on the left side of Figure 5.23. In order to see all the activities and tasks of the process, users can make use of the "All Tasks" action of the toolbar. This action acts as a toggle button, as Figure 5.23 illustrates. When the button is deactivated, the Process view only displays the executable tasks (i.e., the tasks that can be performed in the current state of the process). When the button is activated, all of the tasks of the process are displayed. In this case, non-executable tasks are displayed in red, executable tasks are displayed in green, and the tasks that have already been executed are displayed in blue. The color of the activities depends on their nested tasks and subactivities. An activity is shown in blue if and only if all of its tasks and sub-activities have already been executed. On the other hand, the activity is shown in red if and only if all of its

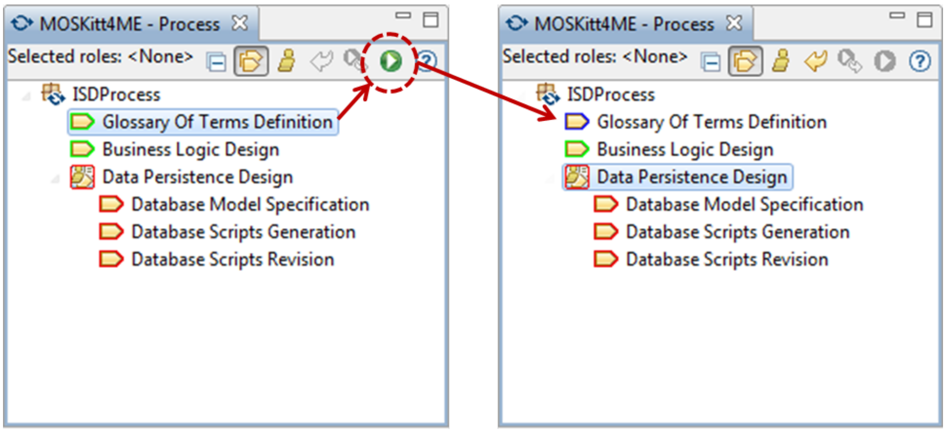

MOSKitt4ME - Process
Selected roles: <None>
ISDProcess
Glossary Of Terms Definition
Business Logic Design
Data Persistence Design
Database Model Specification
Database Scripts Generation
Database Scripts Revision
MOSKitt4ME - Process
Selected roles: <None>
ISDProcess
Glossary Of Terms Definition
Business Logic Design
Data Persistence Design
Database Model Specification
Database Scripts Generation
Database Scripts Revision

FIGURE 5.24: Executing a task in the process view

tasks and sub-activities are non-executable. Otherwise, the activity is shown in green.

Displaying tasks in different colors represents useful guidance for software engineers since it tells them the tasks to be executed based on the current state of the project. In order to execute a specific task, software engineers must double-click the (executable) task in the Process view. When the user carries out this action, different possibilities exist. For instance, if the task is associated to a model transformation, then the project manager of MOSKitt4ME launches the execution of the transformation; or, if the task has an output product that has a graphical editor (or a metamodel) associated to it, then the Project Manager opens a wizard that enables the creation of an empty model. This model will be manipulated by means of the graphical editor (or a default tree-based editor in the case of the metamodel).

When the execution of a task has been requested and the output products are already created, subsequent execution requests for the same task do not have the same effect. Specifically, when the user double-clicks an executable task that has already been started, the project manager opens its output products so that the user can modify them.

Once a task is considered to be finished, the user must manually set the task as executed. This can be performed by means of the "Run" action of the toolbar.

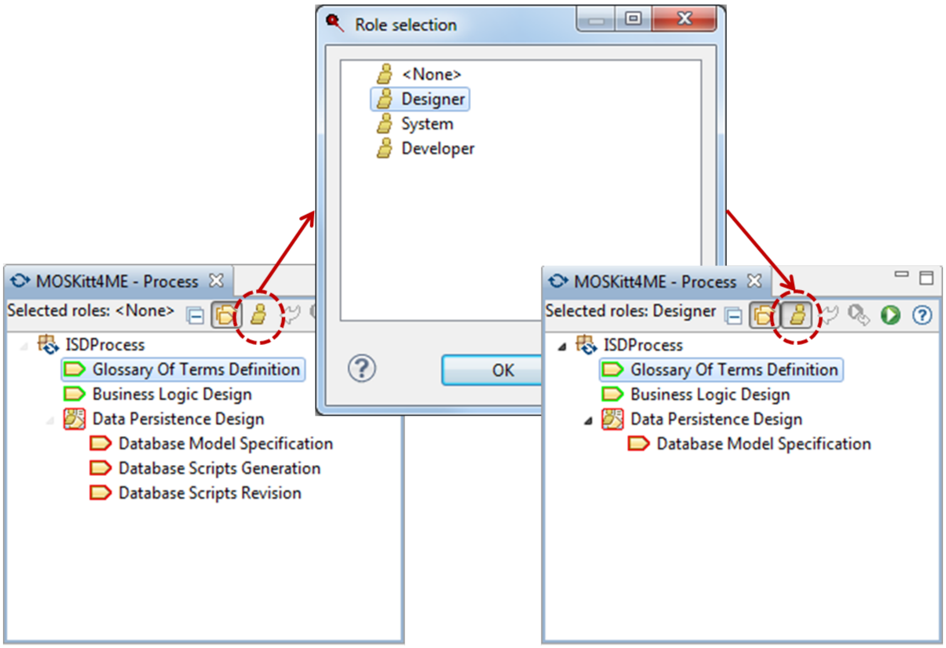


FIGURE 5.25: Selection of roles in the process view

When a task is set as executed, the project manager notifies the Activiti Engine, which takes the process instance to its next state. This is illustrated in Figure 5.24. As the figure shows on the left side, the "Glossary of Terms Definition" task is displayed in green before the user invokes the "Run" action. Then, after the invocation, the task is shown in blue, which indicates that the task has already been finished. This is shown on the right side of the figure.

In addition to the "Run" action, the Process view also provides the "Run Repeatable" action. This action has the same icon as the "Run" action plus an overlay icon in the form of an arrow. The "Run Repeatable" action, which can be found in the toolbar of the Process view, is only enabled for the tasks that were set as "repeatable" during the phase of method design. When the "Run Repeatable" action is invoked for a task, the task is considered again as not started. This means that subsequent execution requests will have the same effect as if the task had never been executed.

The Process view also provides support to task filtering based on the role of the user. In order to select a specific role, users can make use of the “Role Selection” action of the toolbar. Similarly to the “All Tasks” action, the “Role Selection” action acts as a toggle button, as Figure 5.25 illustrates. When the button is deactivated, tasks are not filtered; when it is activated, tasks are filtered based on the roles that are selected by the user. The role selection is performed by means of the “Role Selection” dialog. As an example of use of this dialog, Figure 5.25 illustrates that the user selects the “Designer” role, and, thus, the Process view filters the tasks that are not assigned to this role. Note that the toolbar always shows the roles that are selected by the user.

Finally, the Process view also provides support to returning to previous states of the process by means of the “Undo” action of the toolbar . This action allows the user to go back in the process execution but does not delete the files that have already been created.

Product Explorer

When a project is selected in the MOSKitt Resource Explorer, the Product Explorer view shows a hierarchical representation of the method products that have been created for the project. This hierarchy, as Figure 5.26 illustrates, is based on the elements of type “Domain” and “Product” that are defined in the method model. Specifically, the Product Explorer shows (1) the domains that are defined in the method, (2) the subdomains and products that are contained in these domains, and (3) the files that have been created for each particular product. These files represent instantiations of the products in specific development projects.

Similarly to the Process view, the Product Explorer can also be filtered by role. If one or more roles are selected in the Process view, then the Product Explorer will only show the products that are output of the tasks that are assigned to the selected roles. Thus, the user can easily focus on the products he/she is responsible for.

Guides View

The project manager of MOSKitt4ME also provides the Guides view, which can be opened by means of the “Show View” dialog (available at the “Window” menu).

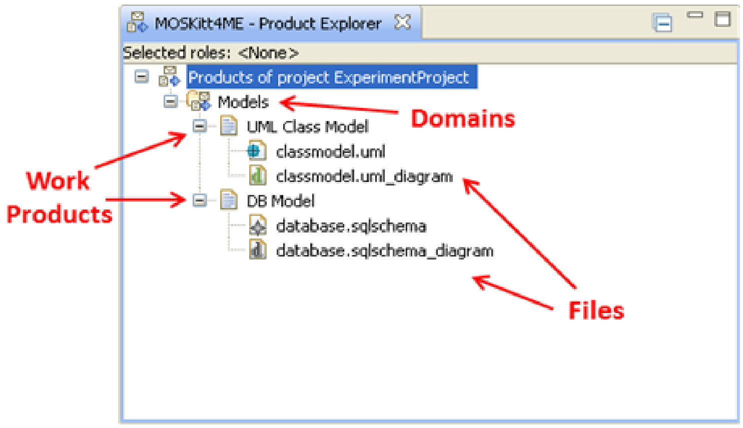


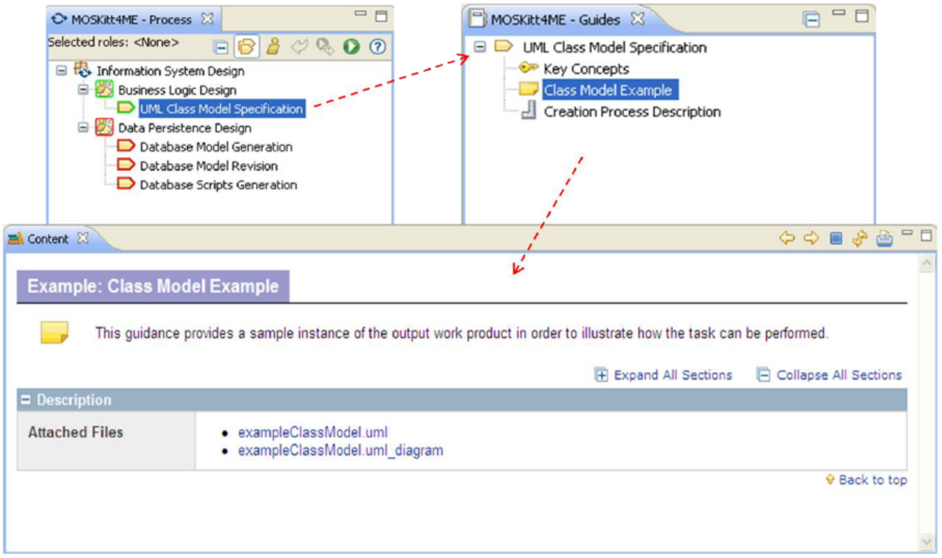
FIGURE 5.26: Product explorer view in MOSKitt4ME

The Guides view – by making use of the method model at runtime – provides software engineers with guidelines on the performance of the method tasks. Specifically, the Guides view shows the elements of type “Guidance” that are associated to the method task that is selected in the Process view. In order to see the content of these guidance elements, the user must double-click them. This action opens the “Content” view of the EPF Composer, which displays an HTML representation of the guidance content.

Figure 5.27 shows how the Guides view operates. Specifically, the figure illustrates that a user selects a task in the Process view. This action updates the Guides view, which shows three elements of type “Guidance”; all of these elements are retrieved from the method model. When the user selects one of these elements, its content is displayed in the Content view.

Help View

As another type of guidelines, the project manager also provides support to the Eclipse dynamic context help. This type of help is shown in the Help view that is provided by Eclipse. Specifically, when a method task is selected in the Process view, the project manager obtains – if any – the guidance fragments that are associated to the task. Then, the guidelines that are contained in these guidance fragments are displayed in the Help view. This is illustrated in Figure 5.28.



Example: Class Model Example

This guidance provides a sample instance of the output work product in order to illustrate how the task can be performed.

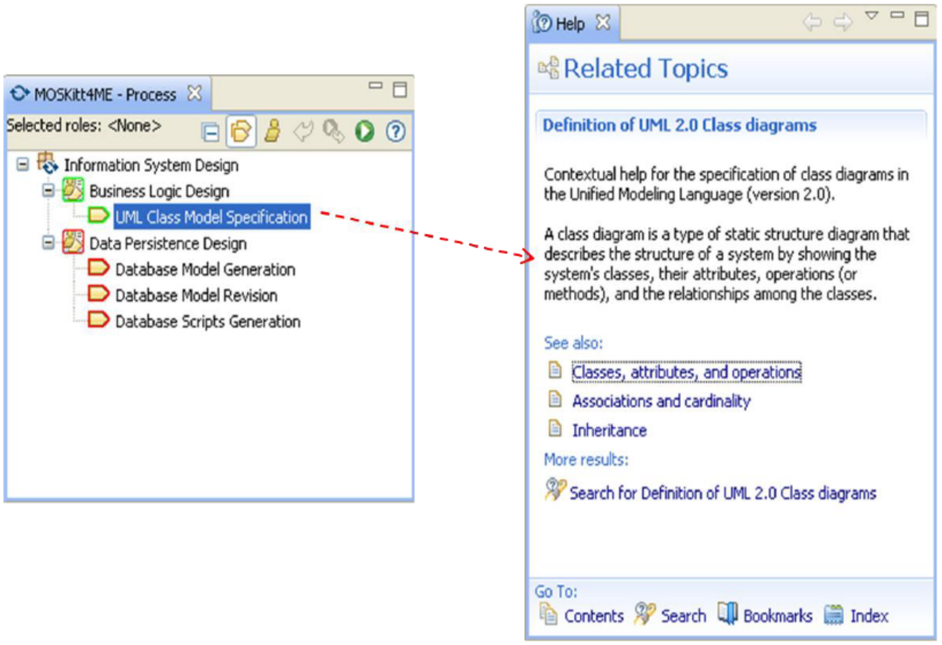
Expand All Sections Collapse All Sections

Description

Attached Files	• exampleClassModel.uml • exampleClassModel.uml_diagram

Back to top

FIGURE 5.27: Guides view in MOSKitt4ME



Related Topics

Definition of UML 2.0 Class diagrams

Contextual help for the specification of class diagrams in the Unified Modeling Language (version 2.0).

A class diagram is a type of static structure diagram that describes the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among the classes.

See also:

- Classes, attributes, and operations
- Associations and cardinality
- Inheritance

More results:

- Search for Definition of UML 2.0 Class diagrams

FIGURE 5.28: Help view in MOSKitt4ME

5.4 Conclusions

This chapter presents MOSKitt4ME, a CAME environment that has been implemented to support our approach for model-driven Method Engineering. To ensure that MOSKitt4ME offered the necessary functionality, we identified a set of functional requirements prior to developing the tool. These requirements were conceived to provide complete support to the lifecycle of Method Engineering. Thus, MOSKitt4ME incorporates components that cover from the initial design of methods (e.g., the method editor) over their implementation (e.g., the CASE generator) to the final method execution (e.g., the project manager).

It is worth mentioning that, by implementing the MOSKitt4ME environment, we show that it is possible to systematize (and partially automate) the design, implementation, and execution of methods. Nonetheless, the benefits of MOSKitt4ME still need to be demonstrated via rigorous evaluation methods. For this reason, we performed a study that evaluates MOSKitt4ME by means of the Technology Acceptance Model (TAM) [45] and the Think Aloud method [48]. This evaluation study is thoroughly described in Chapter 6.

Chapter 6

Evaluation of the Proposal

Despite the potential benefits of CAME technology, this kind of tools have never been widely adopted in industry mainly due to the high complexity of existing Method Engineering approaches [3, 31]. In order to facilitate the use of our CAME environment (MOSKitt4ME), one of the major goals of this thesis was to minimize the complexity of the methodological approach that is introduced in Chapter 4.

In the present chapter, we describe a study that we carried out to evaluate the extent to which we accomplished the above thesis goal. The study that is presented in this chapter evaluates MOSKitt4ME with respect to two quality attributes: perceived usefulness and perceived ease of use. We evaluated perceived ease of use because this attribute represents a subjective measure of complexity [46, 47]. We evaluated perceived usefulness because this attribute is causally affected by perceived ease of use [164], and, for this reason, the usefulness of Method Engineering is often negatively perceived by practitioners (which represents a major obstacle for the success of Method Engineering and CAME technology).

In order to evaluate the perceived usefulness and the perceived ease of use of MOSKitt4ME, we applied the Technology Acceptance Model (TAM) [45]. The subjective results that we obtained by means of the TAM were reinforced by the Think Aloud method [48], which allowed us to assess usefulness and ease of use in an objective manner. Thus, we could analyze not only the users' satisfaction with MOSKitt4ME, but also the users' actual performance during the study and the difficulties that they experienced using our tool.

In summary, the main goal of this chapter is to illustrate that MOSKitt4ME can be positively rated in terms of perceived usefulness and ease of use and that it can also improve the users' performance while posing little difficulty of use. Our positive results contrast with traditional Method Engineering, whose usefulness is often negatively perceived by practitioners and whose complexity remains an unsolved issue. As a collateral benefit of the study, this chapter also illustrates how MOSKitt4ME reduces the complexity of Method Engineering by means of Model-Driven Engineering (MDE) techniques.

The remainder of this chapter is structured as follows. First, Section 6.1 provides background on Method Engineering evaluation. Then, Section 6.2 gives an overview of our evaluation study. Each of the four phases that comprise the study (i.e., the initial design of the study, its execution, the analysis of the collected data, and the presentation of results) are detailed in Sections 6.3, 6.4, 6.5, and 6.6, respectively. Finally, Section 6.7 draws some conclusions about the present chapter.

6.1 Background on Method Engineering Evaluation

In 1996, Tolvanen *et al.* [176] noted that Method Engineering researchers had focused mostly on the theoretical foundations of the discipline and highlighted the need for investigating usability issues such as usefulness or complexity. A similar conclusion was reached in 1997 by Ter Hofstede *et al.* [28], who stated that more empirical research was needed to substantiate the claims associated with the potential benefits of Method Engineering. Despite these demands for more empirical research, two decades later it is still hard to find empirical studies that investigate methods and tools for Method Engineering [11].

One of the few empirical studies that have been conducted in the context of Method Engineering is the work by Sousa *et al.* [177]. This work evaluates the graphical notation of a language for method design: the ISO/IEC 24744 standard

[67]. The main contributions of this work are several suggestions for improving the notation, such as extending the taxonomy of action types. Other empirical studies are those by Kelly *et al.* [178] and Kerzazi *et al.* [179]. The former study evaluates an approach for testing metaCASE environments; this approach is based on an error classification that allows the performance of metamodelers to be measured. The latter study evaluates the usability of two method design tools: EPF Composer and DSL4SPM.

In a more theoretical context, we can find two Method Engineering approaches that take complexity into consideration. In [6], Bajec *et al.* present the Process Configuration Approach (PCA), which was conceived to be simple enough to be adopted by software companies. On the other hand, in [4] Karlsson *et al.* propose the Method for Method Configuration (MMC). The MMC is based on the notion of method component [158], which combines Method Engineering with activity theory to make Method Engineering less cumbersome.

In addition to the above research efforts, which deal with usability issues, we can also find empirical studies that concern other aspects of the Method Engineering discipline. For instance, Qumer *et al.* [180] tested the applicability of an analytical framework for assessing the agility of software development methods, while in [181] Karlsson describes the lessons learned in the evaluation of a wiki-based approach for method tailoring. On the other hand, Seidita *et al.* [182] performed a study where they tested their Method Engineering approach for the construction of multi-agent system design processes.

The analysis of all the aforementioned studies allowed us to identify two important limitations. First, most of the empirical research that has been performed in the Method Engineering field only investigates the method design phase of the Method Engineering lifecycle; thus, the method implementation and execution phases are almost completely neglected. Second, even though some authors take complexity into consideration [4, 6, 28], none of them provide a detailed empirical analysis of the usefulness and ease of use of a Method Engineering approach when it is put into practice by means of a supporting CAME environment. In order to fill these gaps, our study makes a detailed analysis of the usefulness and ease of use of a model-driven Method Engineering approach (MOSKitt4ME) when it is put into practice during three phases of the Method Engineering lifecycle: design, implementation, and execution.

6.2 Overview of the Evaluation Study

In this section, before detailing the evaluation study, we present an overview that summarizes (1) the measures that were used to assess usefulness and ease of use and (2) the experimental process that was followed during the study.

6.2.1 Measures of Usefulness and Ease of Use

In our study, we evaluate usefulness and ease of use by means of two types of measures: subjective and objective. Subjective measures are those which concern the perception of users (e.g., a user's estimation of the time invested in a task), while objective measures are free from human judgement (e.g., the real time that is invested by the user).

The use of two types of measures has two main advantages [183]. First, since each type of measure may lead to different conclusions, obtaining similar results reinforces the evaluation study. Second, the combination of two types of measures provides a more complete picture of the phenomenon that is studied. Nonetheless, the utility of using subjective and objective measures depends on the context of use. In our study, we use two types of measures because we are interested in improving the users' objective performance with MOSKitt4ME and also in making their subjective experience with the tool more satisfying.

Figure 6.1 summarizes the measures of usefulness and ease of use that are employed in our study. As the figure shows, the subjective measures evaluate the users' satisfaction with MOSKitt4ME. Similarly to most usability studies (which use questionnaires to quantify satisfaction [183]), we used two standard questionnaires; specifically, the questionnaires that are defined by the TAM [45]. The TAM is the most widely applied model for evaluating usefulness and ease of use in a subjective manner [47, 184]. This evaluation is done through two measures: *perceived usefulness* and *perceived ease of use*.

On the other hand, the objective measures that are used in our study evaluate the performance of MOSKitt4ME users. Specifically, we measured *task completion time* (which is a measure of efficiency) and *task completeness* (which is a measure of effectiveness). These measures quantify the usefulness of MOSKitt4ME in the

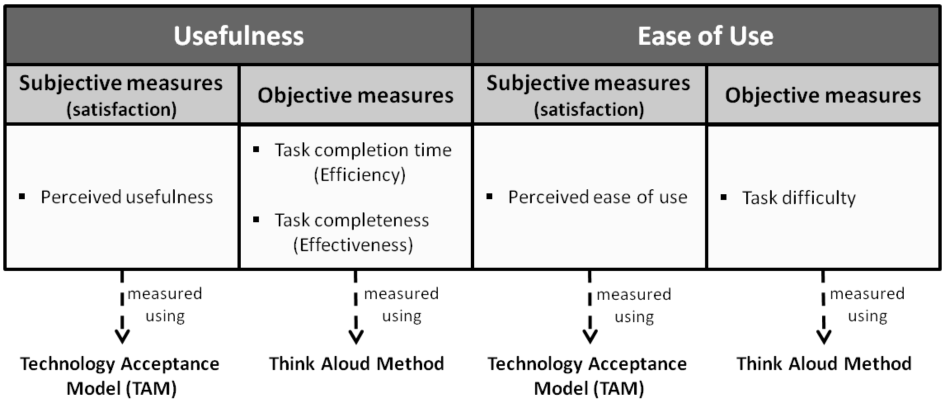

Usefulness
Ease of Use
Subjective measures (satisfaction)
Objective measures
Subjective measures (satisfaction)
Objective measures
Perceived usefulness
Task completion time (Efficiency)
Task completeness (Effectiveness)
Perceived ease of use
Task difficulty
measured using
Technology Acceptance Model (TAM)
Think Aloud Method
Technology Acceptance Model (TAM)
Think Aloud Method

FIGURE 6.1: Measures used in the evaluation study

sense that our tool can be considered useful if it improves performance. Additionally, to evaluate the ease of use of MOSKitt4ME, we measured *task difficulty*. Unlike the other two measures, this measure was tested qualitatively; specifically, it was tested in terms of the challenges (or difficulties) that were faced by the users during the execution of the tasks of the study. These challenges disclose the complexity of MOSKitt4ME, and, consequently, they can be considered to be an objective appraisal of the ease of use of the tool. Identifying these challenges also gives us the opportunity to mitigate them, and, thus, improve MOSKitt4ME to provide better tool support for Method Engineering.

The objective measures of our study were tested through direct observation [185] since this type of evaluation method provides the most in-depth understanding of the phenomenon under study [186]. Of all the methods based on direct observation, we selected the Think Aloud method [48] because it is the most systematic and valid [185, 187]. This method gathers data while a real user-system interaction is taking place, thereby avoiding the problems of interviews and questionnaires. Note that, while interviews and questionnaires are more straightforward, the data that they produce typically represents an incomplete picture of reality [188]. Furthermore, post-use questions to the user may be biased due to the tendency of people to describe their behavior in terms of formal methods (that they acquired during their professional training), while their real actions usually deviate from these methods [48].

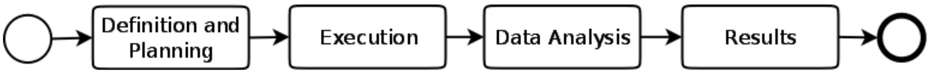


FIGURE 6.2: Phases of the experimental process (adapted from [189])

6.2.2 Experimental Process

For the evaluation of MOSKitt4ME, we followed the guidelines for experimentation in software engineering that are proposed by Wohlin *et al.* in [189]. Based on these guidelines, we followed the process that is shown in Figure 6.2. As the figure shows, this process comprises four sequential phases: (1) definition and planning, (2) execution, (3) data analysis, and (4) results. First, we established the scope of the study (by defining its goal) and its planning (which determines how the study is conducted: design, subjects, research questions, etc.). Second, we executed the study with the subjects in order to collect the data to be analyzed. Third, we analyzed the collected data; we applied procedures and statistical techniques that allowed us to answer the research questions that were proposed for the study. Finally, in the fourth phase, the responses to the research questions were elaborated using the results that were obtained from data analysis. The four phases of the experimental process are detailed in the following sections.

6.3 Definition and Planning

This section details the first phase of the study. In this phase, we defined the goal of the study as well as the research questions, subjects, objects, factors, tasks, context, instrumentation, experimental setup, and validity evaluation.

6.3.1 Goal

The goal of the study is to evaluate two attributes of MOSKitt4ME: usefulness and ease of use. To achieve this goal, we put MOSKitt4ME into practice by means of a set of users that were selected from both academia and industry. Following the template for goal definition that is suggested in [189], the goal of the study can be summarized as follows:

Analyze MOSKitt4ME

For the purpose of evaluation

With respect to usefulness and ease of use

From the point of view of the researcher

In the context of academia and industry

6.3.2 Research Questions

To achieve the goal of the study, we defined four questions that guided the research that was performed in this study. The first two research questions (RQ1 and RQ2) focus on the subjective perception of users; specifically, RQ1 investigates perceived usefulness and RQ2 investigates perceived ease of use.

RQ1. What is the users' perceived usefulness of MOSKitt4ME?

RQ2. What is the users' perceived ease of use of MOSKitt4ME?

The next research questions (RQ3 and RQ4) focus on objective measures; specifically, RQ3 investigates the actual improvement in performance that is provided by MOSKitt4ME and RQ4 explores the actual difficulties that are faced by the users of the tool.

RQ3. To what extent does MOSKitt4ME enhance efficiency and effectiveness?

RQ4. To what extent can MOSKitt4ME be used free from difficulty?

6.3.3 Subjects

Software developers are the population of interest for this study; in practical settings, they are the performers of the methods and they often work as casual method engineers. The study does not require expert developers, but subjects must have basic knowledge in software development methods: design of method models, implementation of tools that support methods, and execution of methods in development projects. Additionally, we require subjects to be familiar with Eclipse and MDE.

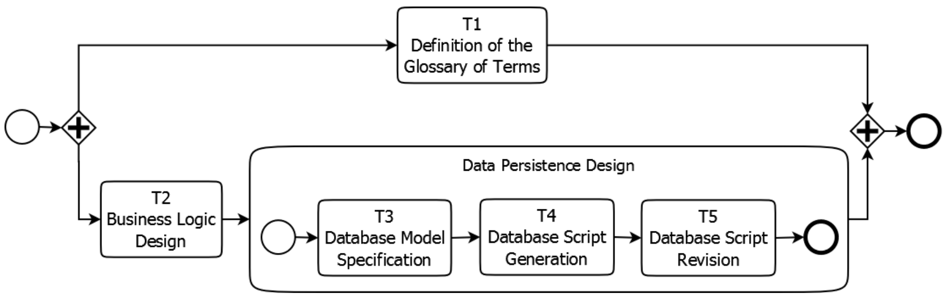

T1
Definition of the Glossary of Terms
T2
Business Logic Design
Data Persistence Design
T3
Database Model Specification
T4
Database Script Generation
T5
Database Script Revision

FIGURE 6.3: The object of the study

TABLE 6.1: Method details

Id	Inputs	Outputs	Roles	Tools
T1	None	Glossary model	Designer	Glossary editor
T2	None	UML 2.0 model	Designer	UML 2.0 editor
T3	UML 2.0 model	Database model	Designer	Database editor
T4	Database model	DDL Script	System	DB2DDL
T5	DDL Script	DDL Script	Developer	None

6.3.4 Object

The object that was selected for the study is a part of gvMétrica[1]: the method that is used at the Valencian Regional Ministry of Infrastructure, Territory, and Environment. The object selection was carried out with a twofold purpose in mind. First, we aimed to find a simple, understandable, and realistic scenario that included enough elements (e.g., tasks, roles, and products) for the complete use of MOSKitt4ME. Thus, subjects could use all of the MOSKitt4ME functionality without being affected by the excessive complexity of the selected object. Second, we aimed to minimize the threat of maturation [189] (see Section 6.3.10.2).

Figure 6.3 shows the object of the study; Table 6.1 contains details about the method tasks. As the figure shows, the first task of the method is to build a glossary model, which defines the terms involved in the system design. This model is built by a designer using a glossary editor. Parallel to the construction of the glossary model, the designer defines the business logic of the system by means of a UML 2.0 editor. Once the UML 2.0 model is finished, the designer defines a

[1] http://www.gvpontis.gva.es/cast/proyectos-integra/

model of the database schema using a database editor; in this task, the UML 2.0 model is used as input. The database model enables the generation of the code that implements the schema in terms of a Data Definition Language (DDL). The DDL code generation is performed by the DB2DDL transformation, which takes the database model as input and produces a DDL script as output. Finally, a developer revises the generated DDL script. The description of the method (as handed out to the subjects during the execution of the study) can be found in the MOSKitt4ME website: `http://users.dsic.upv.es/~mcervera/moskitt4me`.

6.3.5 Factors and Treatments

Our study applies a paired comparison of one factor (*ME approach*) with two treatments (*None* and *MOSKitt4ME*) [189]. In this design, both treatments are applied by all of the subjects of the study. When the subjects apply the *None* treatment, they perform the tasks of the study without using Method Engineering techniques; in contrast, when they apply the *MOSKitt4ME* treatment, they perform the same tasks using MOSKitt4ME. Thus, subjects can compare their subjective perception of using MOSKitt4ME with not using any Method Engineering approach. This design also allows us to evaluate MOSKitt4ME objectively since we can compare the subjects' performance using MOSKitt4ME with their performance without the tool.

6.3.6 Tasks

Based on the selected factor and treatments, we divide the study into two parts – one for each treatment. Below, we describe the tasks to be performed by the subjects in each of these parts. The description of the tasks (as handed out to the subjects during the execution of the study) can be found in the MOSKitt4ME website: `http://users.dsic.upv.es/~mcervera/moskitt4me`.

Treatment 1. ME approach = None. In this part of the study, the subjects perform the tasks without using any tool supporting Method Engineering. This part is divided into two tasks.

- **Task 1.1. Method Design/Implementation.** In this task, we provide the subjects with a printed document that contains the textual and graphical descriptions of the method that is presented in Section 6.3.4. Since the subjects do not have any method editor available, they do not perform the method design; instead, they build a supporting software environment. To enable the subjects to do this task, we give them access to a repository that contains software tools (e.g., graphical editors and model transformations). The challenge lies in manually integrating into the same Eclipse installation only the tools that are strictly necessary to support the method. This can be accomplished by copying the tools into the dropins folder of Eclipse and solving the dependency problems that appear. To identify the dependency problems, the subjects can use the error log that is provided by Eclipse. All of the tools that are required to solve the dependency problems can be found in the repository.
- **Task 1.2. Method Execution.** In this task, the subjects use the software environment that is built in Task 1.1 to run a development project according to the prescribed method. During the course of this project, the subjects must follow the method, executing the tasks in the correct order. To do this, the subjects can only use the printed document as assistance.

Treatment 2. ME approach = MOSKitt4ME. In this part of the study, the subjects perform the same tasks that are described above; however, in this case, the subjects use MOSKitt4ME.

- **Task 2.1. Method Design/Implementation.** Similarly to Task 1.1, we provide the subjects with a printed document that contains the textual and graphical descriptions of the method that is presented in Section 6.3.4. The subjects must use this document to create a model of the method by means of MOSKitt4ME. To enable the definition of the method technical data, we give the subjects access to a repository that contains reusable software tools (e.g., graphical editors and model transformations). When the method model is finished, MOSKitt4ME allows the subjects to automatically obtain the supporting software environment.

- **Task 2.2. Method Execution.** In this task, the subjects use the software environment that is generated in Task 2.1 to run a development project according to the prescribed method. During the course of this project, the subjects must follow the method, performing the tasks in the correct order. This is facilitated by the project manager component that is integrated in the software environment.

6.3.7 Context

The evaluation study was executed in a teaching laboratory of the *Departamento de Sistemas Informáticos y Computación* (DSIC) at the *Universitat Politècnica de València* (UPV). Therefore, the study was executed in an academic environment, not as part of an industrial project. Nonetheless, since our intention was to emulate an industrial environment, the study involved the participation of professionals as well as master/PhD students and postdocs. Additionally, the study addressed part of a real problem: the design, implementation, and execution of an industrial software development method.

6.3.8 Instrumentation

We used five instruments during the execution of the study. These instruments are the following.

Printed document. We provided subjects with a printed document that contains the description of the method that is proposed in Section 6.3.4 and also the tasks of the study. After each task description, the document requests the mental effort that is invested in the task. Mental effort ranges from "very low" (0) to "neutral" (3) to "very high" (6).

Characterization form. This form requests demographic data (such as gender, age, or work status) and it also quantifies the knowledge of the subjects regarding the topics that are covered by the study (see Section C.1). To quantify this knowledge, the characterization form includes twelve multiple-choice questions about the subjects' experience in method modeling, development projects, CASE environments, and technology involved in the study (e.g., Eclipse and MDE). We took experience into consideration since it influences perceived usefulness and perceived ease of use [47].

User acceptance form. The user acceptance form quantifies the subjective perception of the subjects with respect to the usefulness and ease of use of MOSKitt4ME (see Section C.2). We developed this form following the TAM [45], which suggests measuring these attributes by means of two scales of six 7-point Likert items, ranging from “strongly disagree” (0) to “neutral” (3) to “strongly agree” (6).

Interview questions. We elaborated a set of questions to gain further insight into the subjective perception of MOSKitt4ME users (see Section C.3). These questions were divided into two parts. The first part requests the opinion of the subjects with respect to their performance applying the treatments of the study; the second part requests the subjects’ opinion about specific functional aspects of MOSKitt4ME (e.g., the CASE tool generation capabilities).

Physical devices and tools. Following the Think Aloud method [48], we used a webcam to record the subjects’ physical behavior and the uttered thoughts. We also used *HyperCam 3.5* to create screencasts that stored the subjects’ work. The computer that we provided to the subjects was a *HP Spectre XT Pro Ultrabook 13-b000* with *Windows 7 Professional*, *Intel Core i5 1.7GHz*, and *4GB* of *RAM memory*. In this computer, we installed Eclipse, MOSKitt4ME, and the repositories that are required to carry out Tasks 1.1 and 2.1.

6.3.9 Experimental Setup

The process that was followed in the study is shown in Figure 6.4 (A). First, we gathered demographic data by means of the characterization form. Based on this demographic data, we assigned subjects to two groups of equal size and similar average experience. Then, the study was executed as Think Aloud sessions. These sessions were individual; that is, only the experimenter and one subject participated in each session. The groups were used to determine for each subject the treatment to be applied first; thus, we minimized the threat of maturation [189] (see Section 6.3.10.2).

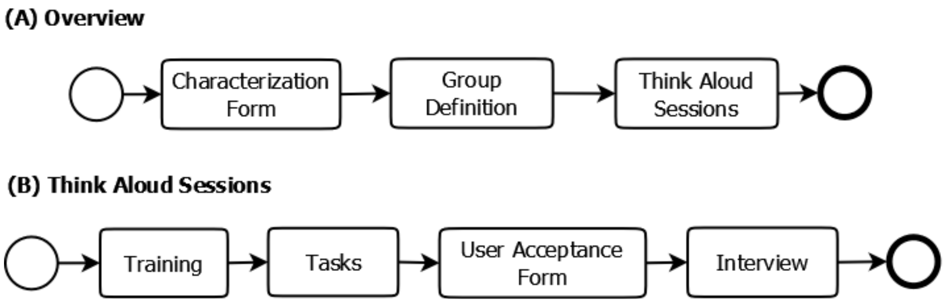


FIGURE 6.4: Experimental setup

The process that was followed in each of the Think Aloud sessions is shown in Figure 6.4 (B). Each session began with a training phase. In this training phase, the experimenter assisted the subject in performing the tasks using a small example method; the experimenter also gave instructions on how to think aloud. After the training phase, the subject performed the tasks using the method that is defined in Section 6.3.4. During the performance of the tasks, the subject was asked to verbalize their thoughts as if he/she was alone in the room. When the subject finished a task, he/she had to specify the mental effort invested. Once all of the tasks were finished, the subject filled out the user acceptance form, and, then, the experimenter conducted the interview.

6.3.10 Validity Evaluation

The results of an evaluation study may be invalidated for different reasons depending on the way the study is conducted. Therefore, it is important to consider the question of validity early in the planning phase so that threats can be minimized. In our study, we considered four types of validity threats: conclusion validity, internal validity, construct validity, and external validity [189].

6.3.10.1 Conclusion Validity

Our study was affected by three threats to conclusion validity. First, our study was threatened by the reliability of the collected measures. Since we video-recorded the Think Aloud sessions, the collection of objective measures was separated from human judgement, and, hence, these measures can be considered to be reliable. We increased the reliability of subjective measures by using scales previously validated in other studies [45]. The second threat appears because the Think Aloud sessions were individual and took place on different dates; thus, the implementation of the treatments may have differed between sessions. To reduce this threat, we replicated the same settings for all of the subjects. Finally, we reduced the random heterogeneity of the subjects via a characterization form that allowed us to evaluate their experience beforehand.

6.3.10.2 Internal Validity

Our study was affected by three threats to internal validity. The first threat is related to the fact that different groups may behave differently (e.g., learning at different rates). We minimized this threat by selecting heterogeneous subjects and placing them in two groups of similar average experience. The second threat is maturation, which implies that subjects may react differently as time passes (e.g., due to boredom or tiredness). To minimize this threat, we designed our study so that one group applied Treatment 1 first and the other group applied Treatment 2 first; additionally, we selected a test object that allowed subjects to finish the tasks in less than two hours. Finally, social threats were avoided because the Think Aloud sessions were individual and the subjects were not allowed to talk to each other about the tasks of the study. Also, since the subjects were not knowledgeable about the treatments to apply, they did not reduce or increase their performance due to excess or lack of motivation.

6.3.10.3 Construct Validity

Our study was affected by two threats to construct validity. The first threat is hypothesis guessing. To reduce this threat, we hid the goal of the study and the mechanisms used to collect data; thus, subjects could focus on the task at hand in the most spontaneous way possible instead of trying to get results that would favour or harm the study. To reduce the second threat, we minimized the effect of the experimenter expectancies; specifically, we reduced the interaction between the experimenter and the subjects to a minimum. During the think aloud sessions, the experimenter only said “keep talking” to remind the subjects to keep thinking aloud.

6.3.10.4 External Validity

Our study was affected by two threats to external validity. The first threat involves the selection of subjects that are not representative of the population of interest. We minimized this threat by selecting software developers from two industrial software companies. The second threat involves having an experimental setting that is not representative of industrial practice. To minimize this threat, we utilized tools that are commonly used in industrial environments (e.g., the Eclipse platform); additionally, the object of the study is part of an industrial method: gvMétrica. Nonetheless, further experimentation is needed to assess how far the results of our study can be generalized to industrial settings and to other types of development methods.

6.4 Execution

This section details the second phase of the experimental process: execution. This phase involves three steps: preparation (where the required material was elaborated and the subjects were chosen), operation (where the selected subjects performed the tasks of the study), and data validation (where we verified that the gathered data was reasonable and that it had been collected correctly).

6.4.1 Preparation

The preparation for the study included the elaboration of the required documents and measurement instruments; that is, the method description, the tasks descriptions, the characterization form, the user acceptance form, and the interview questions. We also prepared the material required for the training phase.

Once the required material was ready, we selected the subjects according to stratified random sampling [189]. Our population comprised two groups: one academic and one industrial. The former group was composed of master/PhD students

Table 6.2: Subjects of the study

Id	Gender	Age	Work Status	Degree
S1	Female	41-55	Professional	Engineer
S2	Female	26-40	Academic	Master
S3	Male	26-40	Academic	PhD
S4	Male	26-40	Professional	Master
S5	Female	26-40	Academic	Master
S6	Male	26-40	Professional	Engineer
S7	Male	26-40	Professional	Engineer
S8	Female	18-25	Academic	Engineer

Table 6.3: Distribution of the subjects

	Group G1				Group G2			
Subjects	S1	S4	S6	S7	S2	S3	S5	S8
Experience	4.33	2.67	2.17	3.25	3.67	3.67	3.42	1.75
Average	3.10				3.12			

and postdocs from the DSIC department; all of them had no relationship with MOSKitt4ME but they worked in the area of software engineering. The latter group comprised software engineers from two valencian companies. The result of the selection is shown in Table 6.2. As the table shows, we selected eight subjects: four males and four females. One subject was under 25 years old; another subject was over 40 years old. In general, their ages ranged from 26 to 40 years old. One of the subjects was a master student (S8), two were PhD students (S2 and S5), and one was a postdoc (S3); the rest were industrial software engineers. We selected eight subjects since small samples are adequate in Think Aloud studies due to the richness and large amount of data that is produced [185, 190].

With respect to the experience level of the subjects, the characterization form revealed that they had low experience in method modeling, medium in development projects and CASE environments, and high in Eclipse and MDE. Based on the subjects' experience (which was measured on a scale from 0 to 6), we evenly distributed them in two groups: G1 and G2. Table 6.3 shows the resulting distribution. The experience level of each group was calculated by averaging the experience level of the subjects pertaining to the group. The experience level of each subject was calculated by averaging the results obtained for the questions of the characterization form.

In addition to the elaboration of material and the gathering of demographic data, the preparation phase also involved the execution of a pre-test, where we simulated a Think Aloud session prior to the actual study. The pre-test allowed us to ensure the feasibility of the general setup and to improve the comprehensibility of the textual documents. The person that was selected for the pre-test did not participate in the actual study; therefore, her results were not considered for analysis.

6.4.2 Operation

We successfully conducted the eight Think Aloud sessions over a two-week period in October 2013. The sessions lasted approximately 2.5 hours on average. To replicate the same settings in all of the sessions, we provided subjects with the same installations of Eclipse and MOSKitt4ME, and these tools were restored to their original state after each session. Additionally, to ensure that the experimental setup was strictly followed by all of the subjects, the experimenter stayed inside the laboratory throughout the duration of the entire sessions. Nonetheless, he only talked to break silences after a fixed interval of 30 seconds. If the subjects needed help, they were allowed to consult the MOSKitt4ME user manual and also the slides that were used by the experimenter in the training phase. The user manual and the slides were printed and handed out to the subjects at the beginning of each session.

As an illustration of a Think Aloud session, Figure 6.5 shows a snapshot of a subject using the software environment that is generated by MOSKitt4ME at the end of Task 2.1 (see Section 6.3.6). As the figure shows, the camera was directed at the subject to give a clear view of the subject's face and hand movements. This facilitated the subsequent interpretation of the verbal data that was produced during the sessions.

6.4.3 Data Validation

As the Think Aloud method suggests [48], only one subject participated in each session, and, thus, we could easily ensure that the experimental setup was strictly followed by all of the subjects. We are also confident that all of the subjects

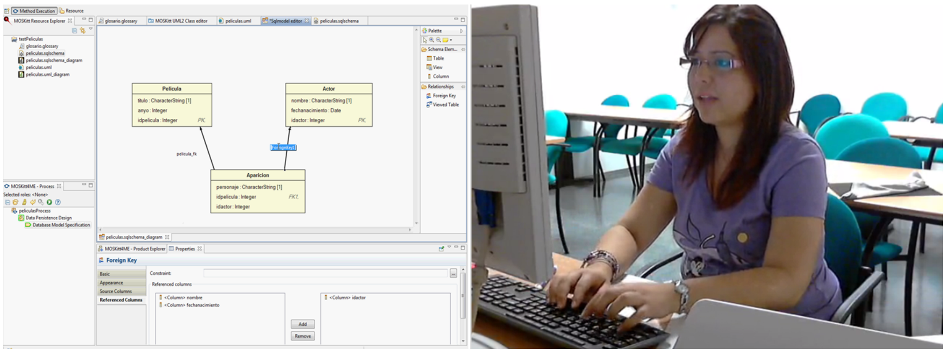


FIGURE 6.5: One of the subjects during a Think Aloud session

understood how to fill in the user acceptance form and how to assess mental effort since we explained these tasks in great detail to all of them.

6.5 Data Analysis

In this section, we describe the third phase of the experimental process, which involves the analysis of the data that is collected during the execution phase. This section is divided into two subsections. One subsection deals with subjective data, which allowed us to answer RQ1 and RQ2; the other subsection deals with objective data, which allowed us to answer RQ3 and RQ4.

Note that this section describes how we carried out the analysis of the data (i.e., the analysis processes that we followed, the calculations that we performed, and the statistical techniques that we applied); we also describe the rationale behind the decisions that were made during data analysis. The results of the analysis are reported in Section 6.6.

6.5.1 Analysis of the Subjective Data

The subjective data that was collected in our study corresponds to: (1) the quantitative feedback obtained by means of the user acceptance form, (2) the qualitative feedback obtained during the interviews, and (3) the mental effort that was reported by the subjects after the execution of the tasks of the study.

6.5.1.1 Quantitative Feedback

We analyzed the responses of the user acceptance form in order to obtain a quantitative view of the subjects' perceived usefulness and ease of use of MOSKitt4ME. To obtain this quantitative view, we considered the numerical values of the responses: from 0 for "Strongly disagree" to 6 for "Strongly agree". Thus, we could calculate the minimum, maximum, and average values for each of the Likert items of the form (and also the total averages combining all of the items). Additionally, we calculated the frequencies of the responses. The frequency of a response is the sum of occurrences of the response divided by the total number of questions. All of this data allowed us to answer RQ1 and RQ2.

6.5.1.2 Qualitative Feedback

In order to reinforce the results obtained for RQ1 (i.e., perceived usefulness), we analyzed the qualitative feedback that was collected during the interviews. This feedback was analyzed in two steps. First, we focused on the first part of the interviews, which deal with the subjects' performance. This part allowed us to determine whether the subjects considered MOSKitt4ME to be useful; that is, whether they believed that MOSKitt4ME improved their performance. Second, we analyzed the second part of the interviews, which deal with the MOSKitt4ME functionality. This part allowed us to assess perceived usefulness with respect to specific functional aspects of MOSkitt4ME (rather than MOSKitt4ME as a whole).

6.5.1.3 Mental Effort

In order to reinforce the results that were obtained for RQ2 (i.e., perceived ease of use), we analyzed the mental effort that was invested by the subjects in the tasks of the study. To this end, we performed Wilcoxon signed-rank tests [189] using *IBM SPSS Statistics 2.0.* The Wilcoxon test is an appropriate technique in our study for two main reasons. First, we have repeated (paired) samples. We have paired samples because the two tasks of the study (i.e., method design/implementation and method execution) were tested twice (with and without MOSKitt4ME). Second, the Wilcoxon test (in contrast to the paired t-test) is a non-parametric technique that does not require the data to be normally distributed, a requirement that was not met in our study. The normality tests that we performed are presented in Section C.4.

By performing the Wilcoxon tests, we were able to verify if there was a significant difference between two datasets: the mental effort invested by the subjects when they did not use MOSKitt4ME (i.e., Treatment 1) and the mental effort invested using our tool (i.e., Treatment 2). The difference between the two treatments should be in line with the results obtained for perceived ease of use; note that, for example, a subject who invests little mental effort using MOSKitt4ME should consider the tool as easy to use.

Specifically, we performed two Wilcoxon signed-rank tests. The first test focused on the method design/implementation, while the second test focused on the method execution. We considered the Wilcoxon tests to be two-tailed and they were performed at a confidence level of 95% ($\alpha = 0.05$). The null hypothesis (H_0) was the same for both tests: the median of differences in mental effort is equal to zero. H_0 can be rejected if $p < \alpha$, where p is the p-value obtained from the Wilcoxon tests.

6.5.2 Analysis of the Objective Data

In contrast to the subjective data, the objective data does not involve human judgement; it is obtained by analyzing the subjects' behavior. In our study, the subjects' behavior was stored in video records and screencasts, which were produced during the Think Aloud sessions; therefore, we extracted the objective data from these sources. This data was used to answer RQ3 and RQ4.

The process that we followed to obtain the objective data is outlined in Figure 6.6. First, we transcribed the Think Aloud sessions; that is, we typed out video records and screencasts as verbatim as possible. Then, we annotated the transcriptions using a coding scheme to obtain Think Aloud protocols. The coding scheme, which was developed in parallel to the protocols, contains codes that define different types of utterances and actions. Thus, the protocols are transcriptions whose utterances and actions are classified as, for instance, suggestions, errors, doubts, or opinions. When the transcriptions were fully annotated, we analyzed the resulting protocols. The four tasks of the process are detailed in the following subsections.

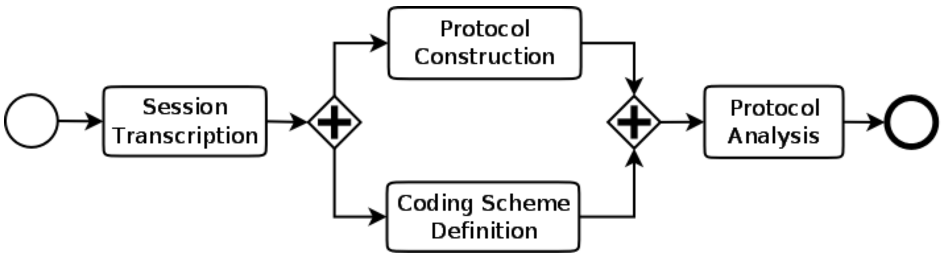


FIGURE 6.6: Data analysis process (adapted from [48])

6.5.2.1 Session Transcription

It is hard to analyze the Think Aloud sessions directly from audio recording and screencasts; for this reason, we transcribed the sessions into text, and, then, we divided this text into segments. The segments of a transcription represent utterances (which are obtained from the video records) and actions (which are obtained from the screencasts). We produced a total of 8 transcriptions, which have 895 segments on average. Each of the segments of these transcriptions stores three items: time, type, and text. *Time* indicates the exact moment of occurrence of the segment within the Think Aloud session. *Type* determines whether the segment is an utterance or an action. Finally, *text* represents the segment content. The content of an utterance is the textual representation of the subject's verbalization; the content of an action is a short description of the action.

6.5.2.2 Coding Scheme Definition

Because it is difficult, and therefore unreliable, to analyze the transcriptions "as is", it is necessary to make a coding scheme to help in the analysis. The coding scheme defines codes that allow the segments of the transcriptions to be categorized, thereby bringing structure to the unstructured data [191].

Similarly to grounded theory [192], we developed the coding scheme applying a bottom-up approach; that is, we analyzed the transcriptions, and, concurrently, we created new codes for each segment that did not fall neatly into the existing coding scheme. Then, we categorized the resulting codes. The final version of the coding scheme includes 90 codes in 7 different categories (see Section C.5). The categories of the coding scheme are the following:

Actions (A). General actions that are performed during the execution of the tasks of the study; for instance, deleting a file.

Tasks (T). Actions that correspond to tasks of the method that is proposed as test object (see Section 6.3.4); for instance, creating a glossary model or revising the DDL script.

Errors (E). Actions that do not adhere to any valid solution for the task at hand; for instance, setting a name incorrectly.

Comments (C). General utterances, such as opinions, suggestions, or doubts; for instance, the utterance "I do not know if I selected the right tool" represents a doubt.

Strategies (S). Utterances or actions whereby subjects express or adopt a plan to achieve a goal. Examples of strategies are trial and error, and postponing an analysis; for instance, the utterance "I think it is better to work this out later" represents postponing an analysis.

Expert Knowledge (EK). Utterances or actions whereby the subjects either suggest that they require further knowledge to carry out a task, show that they have previous knowledge about a task, or gain new knowledge during the performance of a task. For instance, the utterance "I do not need to check this because I am familiar with the tool" reflects that the subject has previous knowledge.

Challenges (CH). Utterances or actions that suggest the presence of a challenge or difficulty. Indicative of challenges can be utterances or actions from the E, C, S, and EK categories. For instance, if a subject applies the "postponing an analysis" strategy, then the subject is probably facing difficulties that he/she decides to work out later. Another example is the following utterance from the EK category: "I require technical details about the tools to make the right choice". This utterance suggests that it is hard to select the right tools to perform the method tasks during the method execution. In addition to utterances or actions from the aforementioned categories, we also consider long periods of silence to be indicative of challenges (since these periods typically represent that the subject is experiencing difficulty).

Table 6.4: Excerpt of a Think Aloud protocol

Time	Type	Text	Code
1:31:50	Action	Looks at method description	A10
1:31:50	Utterance	And now, business logic design	C6
1:31:53	Utterance	It is a UML class diagram	C3
1:31:54	Action	Looks for UML 2.0 editor	A29
1:31:57	Utterance	UML model	C7
1:31:58	Action	Selects the "UML Model" tool	A30, E1, CH2
1:31:59	Utterance	I assume that it is UML model	S1, EK3

6.5.2.3 Protocol Construction

The Think Aloud protocols are the coded versions of the transcriptions; that is, transcriptions that have been annotated with codes from the coding scheme. The coding process should be carried out by people who are not involved in the evaluation study; the researchers that design and execute the study may be too attached to specific hypotheses, and, therefore, they can be biased towards certain outcomes of protocol analysis [48]. To increase the objectivity of our coding process, we selected two researchers that were external to the study and we trained them in the use of the coding scheme. These researchers revised the Think Aloud protocols that we coded; all of the discrepancies were discussed and fixed when agreements were reached.

Table 6.4 shows an excerpt of a protocol; each row represents a different segment. In this example, the subject starts by consulting the method description (A10). After this action, the subject verbalizes the information that is retrieved from the method description (C6) and resolves that he must create a UML class diagram (C3). To this end, he looks for the UML 2.0 editor (A29). When the subject is performing the search, he comes across a tool called "UML Model", reads it (C7), and selects the tool (A30). Since this tool is not the correct choice (E1), we consider selecting the correct tools to be a challenge of the method execution (CH2). Finally, the subject verbalizes that he assumes "UML Model" is the correct tool. He is adopting a "trial and error" strategy (S1), which indicates that he requires further technical knowledge (EK3).

6.5.2.4 Protocol Analysis

Once the protocols were obtained, we analyzed them to measure task completion time, task completeness, and task difficulty (see Section 6.2.1).

Task Completion Time. We used the *Time* column of the protocols to calculate the exact time that was spent by the subjects on the tasks of the study. We applied Wilcoxon tests to analyze the differences between the tasks of Treatment 1 and Treatment 2. Thus, we could determine whether MOSKitt4ME allowed subjects to solve the tasks more quickly.

Task Completeness. In order to calculate task completeness, we used the *Type*, *Text*, and *Code* columns of the protocols. As for the task of method design/implementation, we analyzed the subjects' actions and errors (i.e., the segments of type "Action" and code categories "A" and "E", respectively); this allowed us to determine whether the subjects built a software environment that provided complete support for the method. To determine the completeness of the task of method execution, we analyzed the subjects' errors and also the actions associated to method tasks (i.e., the segments of type "Action" and code category "T"); thus, we could determine the number of method tasks that were successfully performed by the subjects. We consider that a subject successfully performs a task if two conditions are met: (1) the subject obtains the correct output product (i.e., he/she creates a file with the right content and the right file extension) and (2) the subject utilizes the right tool to create the product (e.g., he/she utilizes the glossary editor to create the glossary model).

Task Difficulty. In order to estimate the difficulty of the tasks of the study, we analyzed the protocol segments falling into the "CH" category of the coding scheme. Since these segments represent challenges or difficulties faced by the subjects, they allowed us to determine whether using MOSKitt4ME was a big effort for the subjects or rather it posed little difficulty.

TABLE 6.5: Results for perceived usefulness

Item	Min	Max	Avg
1 - Allows working more quickly	5	6	5.625
2 - Improves job performance	4	6	5.5
3 - Increases productivity	4	6	5.375
4 - Enhances effectiveness	4	6	5.375
5 - Makes work easier	5	6	5.375
6 - Is useful for the job	4	6	5.25
Total Average			5.42

6.6 Results

In this section, we focus on the last phase of the experimental process, which involves the presentation of the results obtained after data analysis. To present these results, we answer the four research questions that are formulated in Section 6.3.2.

6.6.1 Research Question 1

In this section, we answer the following question:

RQ1. What is the users' perceived usefulness of MOSKitt4ME?

After the analysis of the responses of the user acceptance form, we obtained the results that are shown in Table 6.5. The minimum (Min) and maximum (Max) columns indicate that all of the subjects somewhat agreed (4), agreed (5), or strongly agreed (6) about each of the items of the usefulness scale: all of the values fall within the range 4–6. We obtained the best result for the first item (average: 5.625); that is, the subjects expressed a strong belief that tasks can be performed more quickly with MOSKitt4ME. The subjects also positively rated the improvement in job performance (average: 5.5). In general, the subjects agreed that MOSKitt4ME was useful for solving the tasks of the study (total average: 5.42).

In order to provide further insight into perceived usefulness, Figure 6.7 shows a histogram that graphically depicts the distribution of responses of the subjects.

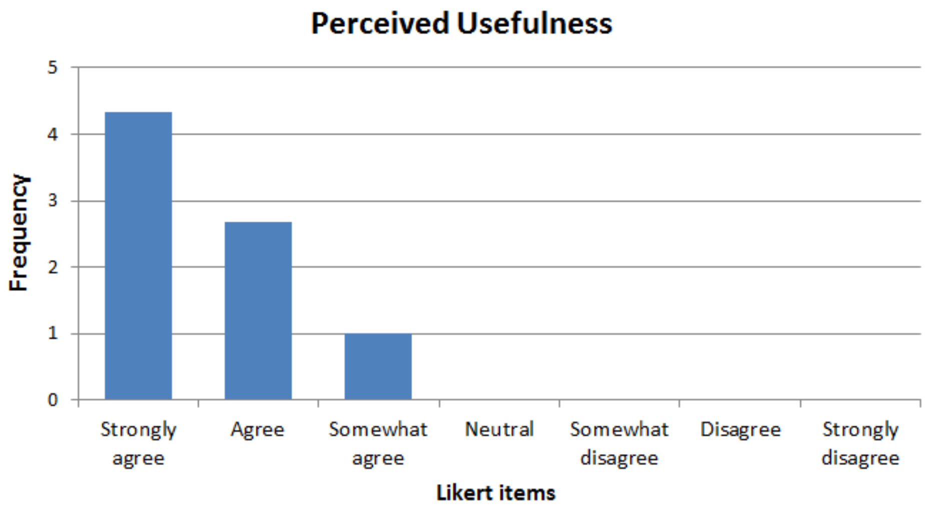


Figure 6.7: Frequencies of responses for perceived usefulness

The horizontal axis of this histogram contains the seven possible responses for the Likert items of the usefulness scale; the vertical axis represents the frequency of occurrence of these responses. The histogram shows that the most common responses were "Strongly agree" and "Agree"; on average, more than 4 subjects selected "Strongly agree" in each of the Likert items and nearly 3 subjects selected "Agree".

These positive results were reinforced by the qualitative feedback that was obtained during the interviews. All of the subjects expressed that MOSKitt4ME was useful since it allowed them to perform the tasks more easily. Most subjects emphasized the method execution; for instance, one subject stated: "*Executing the method with MOSKitt4ME, you click on the tasks and the tools are automatically opened. Without MOSKitt4ME, it is hard to find the right tools in the large set of tools that are offered by Eclipse*", while another subject said: "*When you execute the method with MOSKitt4ME, everything is tool-assisted; for instance, you just double-click on the tasks and the tools are automatically opened. Without MOSKitt4ME, you need additional documentation that tells you what to do next; otherwise, you would lose the awareness of the method you are following*". The usefulness of the CASE generation capabilities was also emphasized by some subjects: "*I would not invest the time needed to implement a CASE environment. Using MOSKitt4ME, I would consider the possibility*". Four subjects also highlighted that MOSKitt4ME facilitates the method design by enabling the definition of methods as models at a high level of abstraction: "*It is so much more user-friendly and intuitive to edit the method using MOSKitt4ME, compared to the textual descriptions that we use in our company.*". Finally, we also found comments that, despite being more general, illustrate the subjects' willingness to accept and use MOSKitt4ME: "*I hate to do work that can be avoided. Knowing the functionality of MOSKitt4ME is possible, I would not want to work differently from now on*".

6.6.2 Research Question 2

In this section, we answer the following question:

RQ2. What is the users' perceived ease of use of MOSKitt4ME?

The results for perceived ease of use are shown in Table 6.6. Compared with perceived usefulness, the results were also positive, but we found more dispersion in them. The minimum (Min) and maximum (Max) columns indicate that all of the subjects considered (4, 5, or 6) MOSKitt4ME to be clear, understandable, flexible, and easy to use; but this did not occur for the other three items of the ease of use scale. We obtained the worst result for the "Controllable" item (average: 2.625); that is, most of the subjects did not consider MOSKitt4ME to be controllable. This means that it was not easy for some subjects to get MOSKitt4ME to do what they wanted it to do. We believe that this result was due to the low level of experience using MOSKitt4ME: we observed that most subjects frequently consulted the user manual. In general, the subjects somewhat agreed that MOSKitt4ME can be used with little difficulty (total average: 4.35).

In order to provide further insight into perceived ease of use, Figure 6.8 shows a histogram that graphically depicts the distribution of responses of the subjects. As this histogram shows, the most common responses were "Agree" and "Somewhat agree"; on average, nearly 4 subjects selected "Agree" in each of the Likert items and more than 2 subjects selected "Somewhat agree".

Table 6.6: Results for perceived ease of use

Item	Min	Max	Avg
1 - Easy to learn	3	6	4.5
2 - Controllable	1	5	2.625
3 - Clear and understandable	4	6	4.75
4 - Flexible	4	6	4.625
5 - Easy to become skillful	3	6	4.75
6 - Easy to use	4	6	4.875
Total Average			4.35

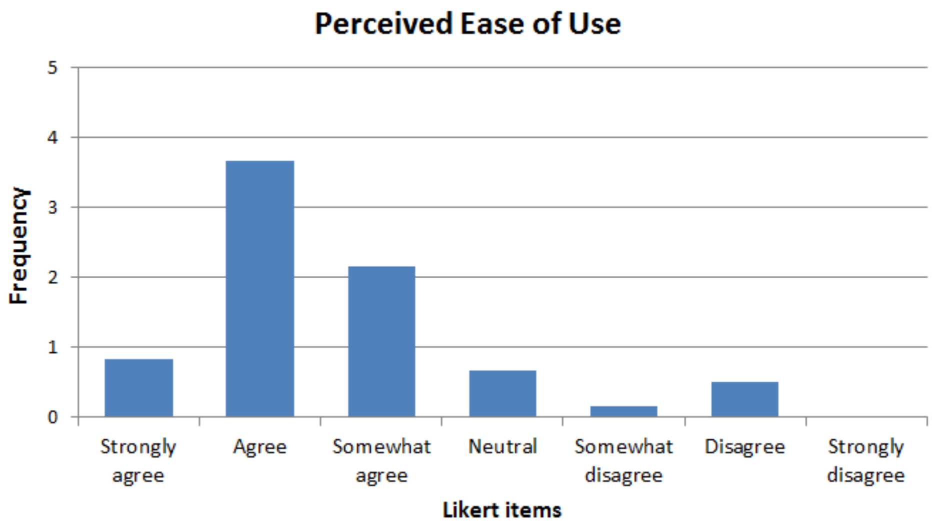


Figure 6.8: Frequencies of responses for perceived ease of use

These positive results were reinforced by the mental effort that was expressed by the subjects after the execution of the tasks of the study. The subjects' mental effort is depicted as a box plot in Figure 6.9. The horizontal axis of this box plot contains the four tasks of the study, while the vertical axis represents mental effort; thus, each box represents, for one specific task, the mental efforts that were invested by the eight subjects of the study. The distribution of the data indicates that the subjects expended less effort executing the method with MOSKitt4ME than executing the method without the aid of our tool. The effort invested in the method design/implementation was low but similar in both approaches. We believe that this similarity was due to the low experience of

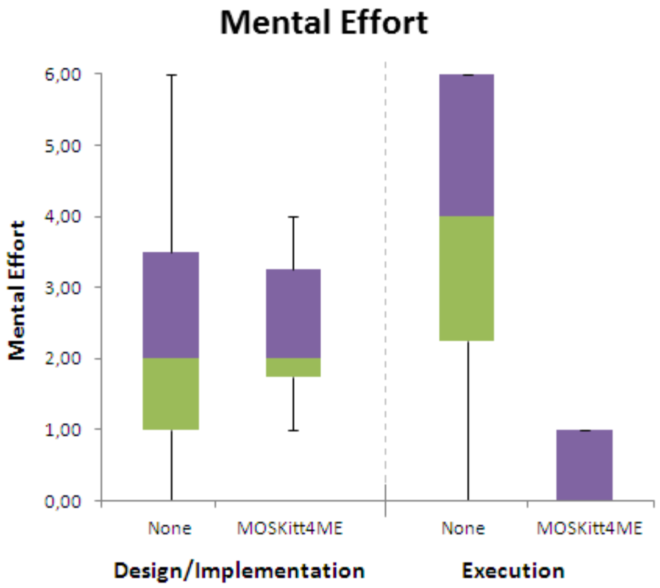


FIGURE 6.9: Mental effort (box plot)

the subjects in method modeling and the high experience in Eclipse. Note that when the subjects were not using MOSKitt4ME, they had to manually configure an Eclipse-based environment, which was easy for some subjects; in contrast, when they used MOSKitt4ME, the environment was obtained automatically but it required the construction of a method model.

To verify whether the differences in mental effort were statistically significant, we performed two Wilcoxon signed-rank tests. In the first test, which focused on the method design/implementation, we obtained $p = 0.931$. Since $\alpha = 0.05$, then $p > \alpha$ and, therefore, we cannot reject H_0. Thus, there is no significant difference in mental effort in the method design/implementation. In the second Wilcoxon test, which focused on the method execution, we obtained $p = 0.027$. In this case, we can reject H_0 since $p < \alpha$. Thus, subjects expended significantly less mental effort when they executed the method with the aid of MOSKitt4ME.

6.6.3 Research Question 3

In this section, we focus on the following question:

RQ3. To what extent does MOSKitt4ME enhance efficiency and effectiveness?

In order to answer this question, we quantified the efficiency and effectiveness of the subjects, and, then, we contrasted the results obtained in the two treatments of the study. Below, we focus on the subjects' efficiency first (Section 6.6.3.1); then, we focus on their effectiveness (Section 6.6.3.2).

6.6.3.1 Efficiency

After protocol analysis, we found that the efficiency of the subjects was affected by their experience level: while some subjects considered some tasks to be trivial, finishing the same tasks took two (or even three) times longer for other subjects. This can be observed in Figure 6.10, which depicts in a boxplot the efficiency of the subjects. The vertical axis of this boxplot represents time; the horizontal axis contains the four tasks of the study. Thus, each box represents, for one specific task, the completion times of the eight subjects of the study. Analyzing the data distribution, we observed that, for example, one subject spent nearly 50 minutes performing the method design/implementation with MOSKitt4ME, while another subject spent less than 20 minutes. In this task, subjects were, in general, more efficient without MOSKitt4ME than using our tool. The goal of this task was to obtain a software environment; thus, MOSKitt4ME failed to reduce the time needed to build this tool. The cause of this result was the high amount of time that some subjects (mainly those with low experience in method modeling) invested building the model of the method, which is mandatory in MOSKitt4ME since the software environment is obtained automatically from this model.

Despite this negative result, it is important to consider that having the method represented as a machine-processable model brings important benefits that are not reaped in the manual approach. Some of these benefits are described in Section 2.2.2. Of these benefits, particularly important is the fact that software environments can execute method models at runtime to assist software engineers during the course of the projects. This benefit became apparent in our study since

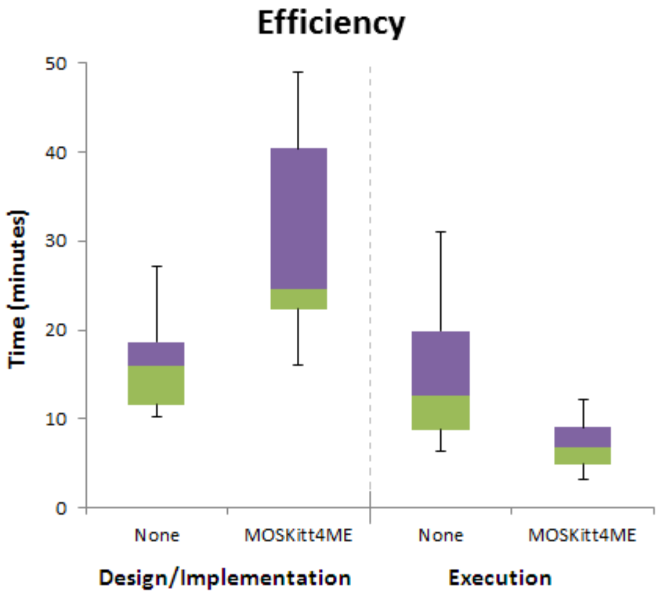


FIGURE 6.10: Efficiency (box plot)

the subjects invested significantly more time during the method execution without MOSKitt4ME than executing the method with the assistance of our tool. This positive result is of particular relevance. Note that, even though the development of the method model is costly in terms of time, this time is only invested once[2]. In contrast, time savings during method execution occur any time a development project is performed. Thus, it seems fair to conclude that MOSKitt4ME improves efficiency if it is applied in multiple projects.

Finally, to verify whether the differences in efficiency were statistically significant, we performed two Wilcoxon signed-rank tests. The first test analyzes the time invested during the method design/implementation; the second test focuses on method execution. In both tests, we obtained $p = 0.012$. Thus, the condition $p < \alpha$ was fulfilled, and, therefore, the differences in efficiency were statistically significant.

[2]In this statement, we do not consider the time that is needed to perform subsequent adaptations of the method.

6.6.3.2 Effectiveness

After the analysis of the protocols, we found that all of the subjects obtained the correct outcome in the task of method design/implementation (effectiveness: 100%). This outcome was obtained in the two treatments of the study; that is, when the subjects used MOSKitt4ME and also when they followed the manual approach. Nonetheless, the manual approach caused severe problems when subjects tried to integrate tools into Eclipse. Since the basic Eclipse installation that we prepared for the study only contained a minimum set of plug-ins, installing new tools "by hand" raised problems of missing software dependencies. Solving these problems was considered by all of the subjects to be complex, tedious, and error-prone. This was reflected in four subjects, who adopted a trial and error strategy; that is, they retrieved random tools from the repository, installed them into Eclipse, and checked whether the tools satisfied the missing dependencies. None of the problems related to software dependencies occurred when the subjects used MOSKitt4ME since our tool automates the construction of the software environment.

In contrast to the method design/implementation (where the subjects were equally effective), we found differences in effectiveness in the method execution. All of the subjects executed the entire method when they were assisted by MOSKitt4ME (effectiveness: 100%), while in the manual approach three subjects abandoned the method execution before completion. These subjects failed to perform T4 because they performed T3 incorrectly: they selected the wrong tool, and, therefore, they created incorrectly the output product of T3. Since this product is the input of the model transformation that supports T4, the error prevented the subjects from launching the transformation and, consequently, from executing T4.

In addition to selecting incorrect tools and creating incorrect products, we also found in the protocols other deviations from the method; for instance, one subject performed T2 and T3 in reverse order and omitted the execution of T5. None of these deviations occurred when the subjects were assisted by MOSKitt4ME.

6.6.4 Research Question 4

In this section, we focus on the following question:

RQ4. To what extent can MOSKitt4ME be used free from difficulty?

By analyzing the protocols, we identified recurring difficulties that were faced by the subjects of the study when they used MOSKitt4ME. All of these difficulties related to two main aspects of the tool: SPEM 2.0 and the technical fragments. Below, we focus on SPEM 2.0 first (Section 6.6.4.1); then, we focus on the technical fragments (Section 6.6.4.2).

6.6.4.1 Difficulty Using SPEM 2.0

Several subjects experienced difficulty understanding the SPEM 2.0 concepts. Specifically, two subjects had problems defining the output products of the tasks: it was not easy for these subjects to distinguish the different kinds of products that are proposed by SPEM 2.0. Additionally, six subjects experienced doubt during the definition of the process: they were uncertain whether or not to make explicit the precedences between the tasks contained in "Data Persistence Design" and the tasks outside this activity. Finally, three subjects had problems distinguishing method content from method process. One of them spent three minutes trying to define an activity within a content package. Another subject took two minutes to realize that process tasks had to be defined by instantiation from content tasks (and not from scratch). The third subject did not realize that the products defined as method content could be instantiated several times; he created the product "DDL Script" twice because it is the output of two different tasks. This was the only error that was made by the subjects as a result of all these minor difficulties.

Some of the subjects that had difficulty with SPEM 2.0 made suggestions during the Think Aloud sessions to facilitate the design of methods with MOSKitt4ME. Examples of these suggestions are: "*I recommend the inclusion of contextual help to assist the user during the specification of the method*" and "*I would rather have a graphical method modeler instead of a tree-based method modeler*".

6.6.4.2 Difficulty Defining Technical Data

Some subjects had difficulty associating method elements and technical fragments, and, in general, understanding the notion of technical fragment. Specifically, one subject stated that he did not understand why (unlike tasks, roles, and products) the method tools had to be defined using a repository. Another subject spent one minute trying to determine the technical fragment to be associated to T5, even though this task must not have any tool associated to it. Another doubt was whether T3 was automatic; note that this task is not automatic because it is supported by a graphical editor (and not by a model transformation). Some subjects also had problems understanding the semantics of some types of technical fragments. All of these doubts caused that three subjects made incorrect associations between method elements and technical fragments; therefore, this is an important aspect of MOSKitt4ME to be improved in the near future.

6.6.5 Discussion

The evaluation study that is presented in this chapter provides valuable insight into the usefulness and ease of use of MOSKitt4ME. Below, we highlight the most relevant aspects of the results that are presented herein:

- The subjective perception that was expressed by the subjects of the study indicates their willingness to accept and use MOSKitt4ME. They perceived MOSKitt4ME to be a useful tool that can improve performance without posing severe difficulties for the users (see RQ1 and RQ2).

- The assistance that is provided by MOSKitt4ME allowed subjects to perform the method execution without deviations, and this led to a significant increase in efficiency and effectiveness (see RQ3). This result suggests that MOSKitt4ME facilitates the use of methods, and, thus, it reduces the distance that typically exists between methods and the real actions that are performed by software engineers [34]. Nonetheless, this benefit is reaped at the expense of the time that is required to build the method model, which may be high in cases of low modeling experience. In order to reduce this time, we plan to enhance MOSKitt4ME in two ways. First, we will increase the starting population of the repository so that it also contains conceptual fragments (specifically, reusable method parts extracted from gvMétrica). This will reduce time by enabling rapid method assembly. Second, we will increase the level of automation of MOSKitt4ME by incorporating variability mechanisms that automate the adaptation of methods and CASE environments.

- The method design is the only phase of the Method Engineering lifecycle where MOSKitt4ME users experienced difficulty during the study (see RQ4). Most of these difficulties were of low severity and were related to the use of SPEM 2.0 and the technical fragments; therefore, these difficulties can be mitigated by enhancing MOSKitt4ME with appropriate assistance for method construction. To do this, we plan to include a wizard that will free users from having to be expert method engineers, allowing them to create method models following a set of intuitive steps. We expect this wizard to also reduce the time invested by the users in the method design phase.

In addition to all of the above findings, our study also had a collateral benefit. Even though it was not the focus of the evaluation, our results suggest that MDE plays a key role in the reduction of Method Engineering complexity that is achieved by MOSKitt4ME. Four subjects strongly believed that (meta)modeling techniques reduce the complexity of the method design phase by enabling the definition of methods as models at a high level of abstraction (see RQ1). This result is in line with one of the most recognized benefits of MDE: the reduction of the complexity of software development by providing higher levels of abstraction that hide platform-specific details [43]. On the other hand, the eight subjects of the study were strongly satisfied with the level of automation of MOSKitt4ME (see RQ1 and RQ3). This level is achieved thanks to model transformations, which reduce the complexity of the method implementation phase by automating the process of CASE environment construction. This is in line with another benefit of MDE: the reduction of complexity by means of the automation of labor-intensive and error-prone tasks [43]. Finally, all of the subjects experienced a significant improvement in efficiency and effectiveness during the method execution phase (see RQ3). This result is a strong indicator of the reduction of complexity that is achieved by using the method model at runtime. Thus, the modeling effort made at design time is not only useful for automating the CASE environment construction, but it can also assist software engineers during the process of software development.

The reduction of complexity that is achieved by MOSKitt4ME makes us believe that MDE may represent a promising alternative to turn Method Engineering into a practical reality. This belief is strengthened by a recent study of the state of practice in MDE [97]; this study concludes that MDE is more widespread than commonly believed and works best when it is applied in narrow and tight domains (such as Method Engineering).

6.7 Conclusions

This chapter presents a study that evaluates the usefulness and ease of use of MOSKitt4ME. Our motivation is to demonstrate that MOSKitt4ME mitigates an important problem of traditional Method Engineering: its high complexity. To achieve this goal, the study evaluates MOSKitt4ME using the TAM and the Think Aloud method. While the TAM enables the evaluation of the subjective perception of users, the Think Aloud method allows us to evaluate the tool objectively.

The results of the study are encouraging. All of the subjects either somewhat agreed, agreed, or strongly agreed about each of the items of the usefulness scale (see RQ1); we also obtained positive results for perceived ease of use, even though we found more disparate opinions (see RQ2). These subjective results were reinforced by an increase in efficiency and effectiveness (see RQ3) as well as by the little difficulty that was experienced by the subjects of the study (see RQ4). We believe that these results were obtained thanks to the use of MDE techniques (such as metamodeling, model transformations, and models at runtime), which reduce the complexity of three phases of the Method Engineering lifecycle: design, implementation, and execution.

In contrast to these positive findings, we also found several challenges that are inherent to MOSKitt4ME usage (see RQ4). With the aim of providing better tool support for model-driven Method Engineering, we will address these challenges in the near future. For instance, as Section 6.6.5 describes, we will enhance the repository of MOSKitt4ME, incorporate support for variability, and include a wizard that enables guided model creation. The main goal of these enhancements is to facilitate the design of methods in MOSKitt4ME.

Chapter 7

Conclusions and Future Work

The present thesis introduces a tool-supported methodological approach that is aimed at method/software engineers that need to carry out the design, implementation, and execution of software development methods. Facing the development of the thesis work from a model-driven perspective allowed us to provide important contributions in the field of Method Engineering. These contributions were presented in relevant international conferences and journals; thus, we expect that our work is properly disseminated not only among researchers from the Method Engineering community, but also among other communities such as Software Engineering, Information Systems, and Conceptual Modeling. It is also important to highlight that the research line in which this work is aligned is by no means completed here. Further work will be undertaken in the near future in order to complement and extend the contributions that have been achieved during the development of the thesis work.

The remainder of this chapter is structured as follows. First, Section 7.1 summarizes the contributions that are provided by this thesis. Then, Section 7.2 describes the publications that emerged during the development of the thesis work. Section 7.3 presents the degree projects and master theses that were co-directed within the context of the present PhD thesis. Finally, Section 7.4 outlines future research directions. These research directions are in line with the limitations of the present work.

7.1 Summary of Contributions of the Thesis

In this thesis, we propose a methodological approach for Method Engineering. This methodological approach offers four novel contributions that go beyond those of traditional Method Engineering approaches. These four contributions (each of which relates to one of the four research questions that are formulated in Section 1.2) can be summarized as follows:

Contribution 1. We show how Model-Driven Engineering (MDE) techniques (such as metamodeling, model transformations, and models at runtime), along with an intensive use of reusability principles, can **reduce the complexity** of a Method Engineering approach.

- By applying *(meta)modeling* techniques, our approach enables the definition of methods as models at a high level of abstraction; thus, our approach reduces the complexity of the method design phase of the Method Engineering lifecycle.
- In our approach, *model transformations* reduce the complexity of the method implementation phase by automating the process of CASE environment construction.
- By using *method models at runtime*, our approach reduces the complexity of the method execution phase; thus, the modeling effort that is made at design time is not only useful for automating the CASE environment construction, but it can also assist software engineers during the entire process of software development.
- In order to foster reusability, we define a *taxonomy of method fragments*. This taxonomy proposes two main types of reusable assets: conceptual fragments and technical fragments. While conceptual fragments alleviate the complexity of the method design phase by enabling rapid method assembly, technical fragments enable a high level of automation in the process of CASE environment construction.

- In order to assess the actual reduction in Method Engineering complexity, we performed an evaluation study that put our methodological approach to practical use. Overall, the results of this evaluation study were favorable. These positive results contrast with traditional Method Engineering, whose complexity still remains an unsolved issue.

Contribution 2. Our approach paves the way for method/software engineers that need to carry out the **design, implementation, and execution of software development methods**. This is unlike traditional Method Engineering approaches, which, in general, only support one of these phases. In order to equally cover the three phases, our approach considers method models as first-class citizens of the entire Method Engineering lifecycle; that is, our approach not only employs method models during the phase of method design but also during the phases of implementation and execution.

- During the *method implementation* phase, method models are used as input of model transformations that automate the construction of the supporting software environments; thus, we bridge the gap between the high-level concepts that are used at method design and the technical details of the method implementations. To bridge this gap, our approach requires a step – the method configuration (see Section 4.2.3.2) – where method engineers connect method concepts with their technical counterparts (i.e., technical fragments).
- During the *method execution* phase, method models (which are created at design time) are used during runtime for driving the behavior of the CASE environments; thus, we bring method models to the runtime context, thereby increasing their value in terms of the functionality that they can deliver.

Contribution 3. We show how Process Modeling Languages can be used in the context of Method Engineering to enable the **specification and execution of the process part of methods without neglecting product support**. This is in contrast to existing Method Engineering approaches, which are mostly product-oriented (i.e., they cover the product part of methods but fail to provide complete support for the process part). In order to equally support the product and process parts of methods, our approach defines a DSL that takes into consideration both the product-related and the process-related aspects of Method Engineering.

- We define a *DSL for the conceptual specification of methods*. This DSL combines concepts from the SPEM 2.0 and BPMN 2.0 standards, and, thus, it reaps benefits from both languages; for instance, SPEM 2.0 provides concepts that are suitable for method modeling, while BPMN 2.0 offers richer workflow-related primitives. The product part of methods is represented in our DSL by means of concepts such as *Product*, *Domain*, and *Tool*. The process part is represented by concepts such as *Task*, *Activity*, and *Role*.

Contribution 4. We provide **MOSKitt4ME: a supporting software infrastructure for our model-driven Method Engineering approach**. This software infrastructure comprises a technology-independent architecture as well as a CAME environment that implements the architecture in the context of Eclipse.

- We define a *technology-independent architecture* that is composed of two main parts: (1) the software components that support the design of methods and the generation of CASE environments (CAME part); and (2) the software components that support method execution (CASE part). Prior to the definition of the architecture, we identified a set of functional requirements.
- We provide a *CAME environment* that implements our software architecture, and, thus, it provides complete support to our methodological approach. This CAME environment allows us to offer the following additional contributions:
  - We demonstrate that it is possible to systematize (and partially automate) the design, implementation, and execution of methods. This is in accordance with one of the principles of design science, which states that the instantiation of a design artifact (e.g., our methodological approach) demonstrates its feasibility [50]. This is called "proof by construction". We reinforced the feasibility of our approach by using the CAME environment to solve an industrial problem: the design of gvMétrica and the generation of a software environment that supports part of the method (see Appendix B).

– We show that our CAME environment can be positively rated in terms of perceived usefulness and ease of use, and that it can also improve the users' performance while posing little difficulty of use. This contribution is achieved via an evaluation study where we collected empirical data by means of questionnaires and direct observation.

7.2 Publications

The contributions that are discussed in Section 7.1 were published in distinct peer-reviewed forums. Our main goal was to validate our work in top conferences and journals within the fields of Method Engineering, Model-Driven Engineering, and Software Engineering. The result was a total of 6 publications. We list these publications in Sections 7.2.1 (conferences and workshops) and 7.2.2 (international journals). For each publication, we briefly describe below the part of the thesis that is addressed by the publication. The author position indicates the degree of contribution that is made by the author of this PhD thesis. Section 7.2.3 provides information about the relevance of some of the conferences and journals where we published our work.

7.2.1 Conferences and Workshops

This section lists in chronological order our publications in international conferences and workshops. Specifically, we published three papers in international conferences (number: 1, 3, and 4) and one paper in a national workshop (number: 2).

1. **Mario Cervera**, Manoli Albert, Victoria Torres, Vicente Pelechano: *A Methodological Framework and Software Infrastructure for the Construction of Software Production Methods.* In proceedings of the 4th International Conference on Software Process (ICSP), vol. 6195, pp. 112–125, Springer-Verlag Berlin Heidelberg, 2010.

The above publication provides a general overview of our methodological approach; it also outlines preliminary ideas about the supporting software infrastructure.

2. **Mario Cervera**, Manoli Albert, Victoria Torres, Vicente Pelechano: *A Technological Framework to support Model Driven Method Engineering.* In Actas del Taller sobre Desarrollo de Software Dirigido por Modelos (DSDM), celebrado junto a las Jornadas de Ingeniería del Software y Bases de Datos (JISBD), vol. 2, pp. 47–56, 2010.

The above publication focuses on a specific phase of our methodological approach: method implementation.

3. **Mario Cervera**, Manoli Albert, Victoria Torres, Vicente Pelechano: *Turning Method Engineering Support into Reality.* In Proceedings of the 4th IFIP WG 8.1 Working Conference on Method Engineering (ME), vol. 351, pp. 138–152, Springer Berlin Heidelberg, 2011.

The above publication focuses on the second contribution of this thesis (see Section 7.1).

4. **Mario Cervera**, Manoli Albert, Victoria Torres, Vicente Pelechano: *The MOSKitt4ME Approach: Providing Process Support in a Method Engineering Context.* In Proceedings of the 31st International Conference on Conceptual Modeling (ER), vol. 7532, pp. 228–241, Springer-Verlag Berlin Heidelberg, 2012.

The above publication focuses on the third contribution of this thesis (see Section 7.1).

7.2.2 International Journals

This section lists in chronological order our publications in international journals.

5. **Mario Cervera**, Manoli Albert, Victoria Torres, Vicente Pelechano: *A Model-Driven Approach for the Design and Implementation of Software Development Methods*. International Journal of Information System Modeling and Design (IJISMD), vol. 3(4), pp. 86–103, IGI Global, 2012.

The above publication was prepared for an special issue called "New Trends in Situational Method Engineering". The paper represents an extension of the paper that was published in ME 2011.

6. **Mario Cervera**, Manoli Albert, Victoria Torres, Vicente Pelechano: *On the Usefulness and Ease of Use of a Model-Driven Method Engineering Approach*. Information Systems, vol. 50, pp. 36–50, Elsevier, 2015

The above publication focuses on the first contribution of this thesis (see Section 7.1).

7.2.3 Relevance of the Publications

This section provides information about the quality level of the most relevant conferences and journals where we published our work. For conferences, we use the ranking that is provided by the Computing Research and Education Association of Australasia (CORE)[1]. On the other hand, for journals, we use the ranking that is provided by the Journal Citation Reports (JCR)[2]. Of the six conferences/journals where our publications were accepted, three of them are included in one of these international rankings. These conferences/journals are ICSP, ER, and Information Systems.

ICSP. The International Conference on Software Process provides a forum for researchers and industrial practitioners to exchange new research results, experiences, and findings in the area of software and system process modeling and management.

ICSP is categorized as a CORE A conference. According to the CORE website, conferences of this category are "excellent conferences and highly respected in a discipline area".

[1] http://www.core.edu.au/
[2] http://thomsonreuters.com/journal-citation-reports.html

ER. The International Conference on Conceptual Modeling is a leading forum for presenting and discussing research and applications in which the major emphasis is on conceptual modeling. Topics of interest include theories of concepts and ontologies underlying conceptual modeling, methods and tools for developing and communicating conceptual models, and techniques for transforming conceptual models into effective implementations.

ER is categorized as a CORE A conference. In 2012, the acceptance rate was 17% (24 papers out of 141 submissions). According to the Conference Ranking in Computer Science (CSCR)[3], the Estimated Impact of Conference (EIC) was 0.90 (measured from 0 to 1), which situates ER in the 11th position out of 636 listed conferences.

Information Systems. The Information Systems journal publishes articles concerning the design and implementation of languages, data models, process models, algorithms, software and hardware for information systems.

According to the JCR, this journal has a 2014 Impact Factor of 1.456 and a 5-Year Impact Factor of 1.618. It is positioned in the second quartile (Q2) and ranked 46th out of 139 journals in the "Computer Science, Information Systems" category.

In addition to ICSP, ER, and Information Systems, it is also relevant to highlight our publication in ME 2011. Even though ME is not listed in the CORE rankings, it represents the most important international conference in the area of Method Engineering. Therefore, our publication in ME 2011 allowed us to disseminate our results within a community that includes many researchers working in the field that is the focus of this thesis.

7.3 Co-directed Projects

In addition to the publications that are listed in Section 7.2, further results of this thesis include a degree project and a master thesis, which were carried out to explore some ideas that are related to our work. The degree project and the master thesis, which were co-directed by the author of the present PhD thesis, are the following:

[3] http://aholab.ehu.es/users/derro/documents/CSConfRank.pdf Accessed 03-2015

1. *Especificación de procesos en la herramienta CAME MOSKitt4ME y construcción de un marco de evaluación para herramientas CAME y CASE.* Alicia Jaén Bielsa. September 2011. Degree project.

2. *CMMI aplicado a entornos de desarrollo software: El caso de MOSKitt4ME.* Edith Elsa Díaz Aspilcueta. September 2013. Master thesis.

7.4 Future Work

The research that is presented in this thesis is not a closed work; it can be improved and extended in several ways. The following subsections summarize the research directions that are planned for the near future. The main goal of this future work will be to overcome some of the limitations of the work that has been developed thus far.

7.4.1 Automated Production of Situational Methods

The methodological approach that is presented in this thesis supports the design, implementation, and execution of methods; however, it does not facilitate the work of method engineers when they need to obtain methods that are adapted to the characteristics of specific development projects. The methods that take into consideration the situation where they will be applied are called situational methods [26].

In order to reduce the effort that is required to obtain situational methods, we plan to enhance our Method Engineering approach with Software Product Line Engineering (SPLE)[4] techniques [193]. The general idea is that, similarly to a product family (or product line), a set of related methods can be decomposed into reusable assets (i.e., method fragments) and organized as a **method family** [194, 195]. Organizing methods as a method family would allow method engineers to exploit commonality and manage variability among method fragments with the aim of automating the production of new methods. Thus, we believe that method families can significantly reduce (through an increased level of automation and

[4] http://www.softwareproductlines.com/

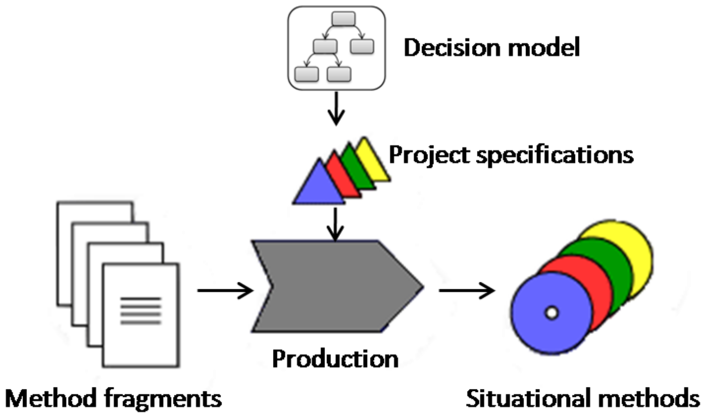


FIGURE 7.1: Basic concepts of a method family

reusability) the effort that is required to obtain methods that are adapted to the characteristics of the projects at hand.

Figure 7.1 graphically depicts the basic elements that come into play in a method family. As the figure shows, a method family supports the creation of a set of *situational methods* from a shared set of *method fragments* using a common means of *production*. Each situational method of the method family is uniquely defined by a *project specification*, which represents choices made in a *decision model*. The decision model defines the whole set of project characteristics (and the relationships between them) that determine all the possible situational methods of the method family. Thus, when a new project starts, method engineers must select (in the decision model) a specific set of characteristics based on this project; then, according to the selected set of characteristics (i.e., according to the project specification), a new situational method is automatically produced by means of the assembly and integration of method fragments.

In order to illustrate how the specification of projects could be performed in MOSKitt4ME, the left part of Figure 7.2 shows a small example of decision model. This decision model is represented as a feature model in terms of the MOSKitt Feature Model Editor. Each of the features of the feature model represents a project characteristic, being the features close to the root more general than the features close to the leaves, which are more specific. Thus, the specification of

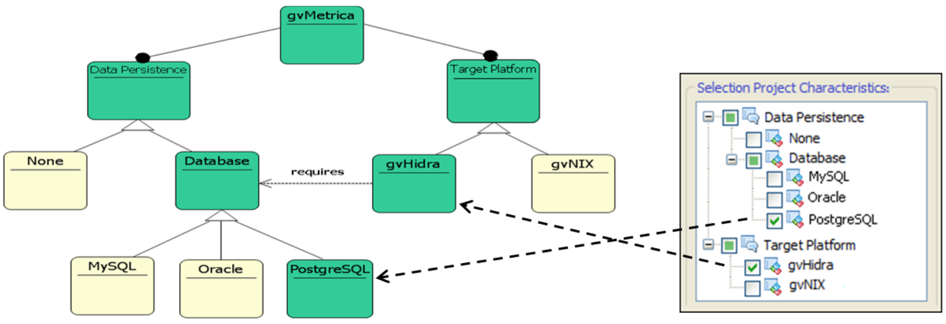


FIGURE 7.2: Example of project specification in MOSKitt4ME

projects could be performed in MOSKitt4ME by means of a graphical component (see right part of Figure 7.2) that displayed as checkboxes all the project characteristics that are available in the feature model. When the user activated a checkbox, then then corresponding feature would be selected. In the example of Figure 7.2, the user is specifying that the project will involve the development of a software application where (1) the data will be persisted in a PostgreSQL database and (2) the target platform will be gvHidra[5]. According to these project characteristics, MOSKitt4ME would automatically produce a method that is adapted for the project; that is, a situational method. This situational method could include, for example, a task called "User Interface Design for gvHidra" and a product called "PostgreSQL database model". This task and this product represent method elements that are adapted to the specific project characteristics that have been selected by the user. If the user had selected, for example, the "None" characteristic (instead of "PostgreSQL"), then the database model would have been omitted from the method.

Note that supporting the automated production of situational methods would be a significant step forward in our Method Engineering approach. Since the method implementation phase is automatic, we would be able to obtain not only situational methods but also customized software environments directly from the project characteristics that are specified by method engineers.

[5] http://www.gvpontis.gva.es/cast/gvhidra-herramienta/

7.4.2 Method as a Service (MaaS)

Recently, there has been a rise in Service-Oriented Architectures (SOA), which has led to a new focus within the Method Engineering community [26]. Rolland elaborates in [10] the idea of applying a service-oriented approach to Method Engineering and defines the concepts of Method as a Service (MaaS) and Method-Oriented Architecture (MOA). Despite its novelty, this idea has only been explored in a limited way thus far. Two examples of research efforts that have been undertaken in this context are those by Gholami *et al.* [138] and Cauvet *et al.* [196, 197]. While the former developed a set of service-oriented method fragments for their inclusion in the OPF repository, the latter investigated how to build methods by discovering, adapting, and dynamically composing method services.

The above approaches show that, up to now, the notion of service has been considered in the Method Engineering field mostly for method design (specifically, as a new type of building block, called method service, whereby methods are built by service composition). Thus, some important aspects remain unexplored. For instance, there is no Method Engineering approach that defines how to obtain integrated software environments that support this new type of service-oriented methods. In the words of Rolland [10]: “a platform permitting an easy execution of method service compositions is missing today”. Another limitation is that there is no Method Engineering approach that allows method engineers to define how method services must be orchestrated during the method execution.

In order to fill these gaps, we plan to incorporate the notion of service to our Method Engineering approach. Thus, the method design phase of our approach will be performed by means of the composition of method services, while the method implementation phase will obtain software environments that enable the execution of these method service compositions. The environments that are obtained during the method implementation phase will be based on the notion of Software as a Service (SaaS) [198]; thus, they will support the creation and manipulation of the method products by means of the invocation of services that are dynamically discovered at runtime. The environments will also support service orchestration through the execution of the process part of the methods (which will be defined in BPMN 2.0) by means of an orchestration engine.

7.4.3 Megamodeling in Method Engineering

During the last decade, there has been an increase in methods that advocate the use of MDE as the means to carry out software development. An example of these methods is gvMétrica. In the projects where this type of methods are applied, models do not represent mere documentation; rather, they embody primary artifacts that drive the construction of software systems. In order for models to become first-class entities of the development process, they must take into consideration all (or, at least, many) of the aspects that concern the software system to be developed (i.e., its requirements, its software architecture, its graphical user interface, the structure of the data that it manipulates, the services that it offers to other systems, etc.). For this reason, it is common to find a large number of interrelated models (expressed by means of many different DSLs) taking part in development projects [199]. This reality is reflected in industrial CASE environments, such as MOSKitt, which support the creation of many different types of models. For instance, MOSKitt supports, among others, UML (activity, class, state machine, sequence, use case, and profile), BPMN, UIM, Sketcher, Dashboard, WBS, and Sqlchema.

The coexistence of many different models within the same development project may cause significant problems. For instance, it may hinder (1) the overall understanding of the system that is under development and also (2) the communication between the people that is involved in the project (which are two of the most recognized benefits of MDE); furthermore, it may lead to (3) inconsistencies between the models and broken references. To avoid all of these problems, CASE environments must facilitate the work of software engineers by offering effective mechanisms for managing and coordinating interrelated models and DSLs. Examples of the facilities that can be offered by CASE environments are high-level specification of constraints between the models, storage of metadata, search of models based on this metadata, storage of traceability information, and graphical visualization of different types of relationships between the models.

Even though the above functionality may bring significant benefits, its development, in contrast, adds further complexity to the process of building CASE environments that are adapted to the context of use – one of the main goals of Method Engineering. To mitigate this problem, it is necessary to define a systematic solution that enables the construction of CASE environments that provide effective

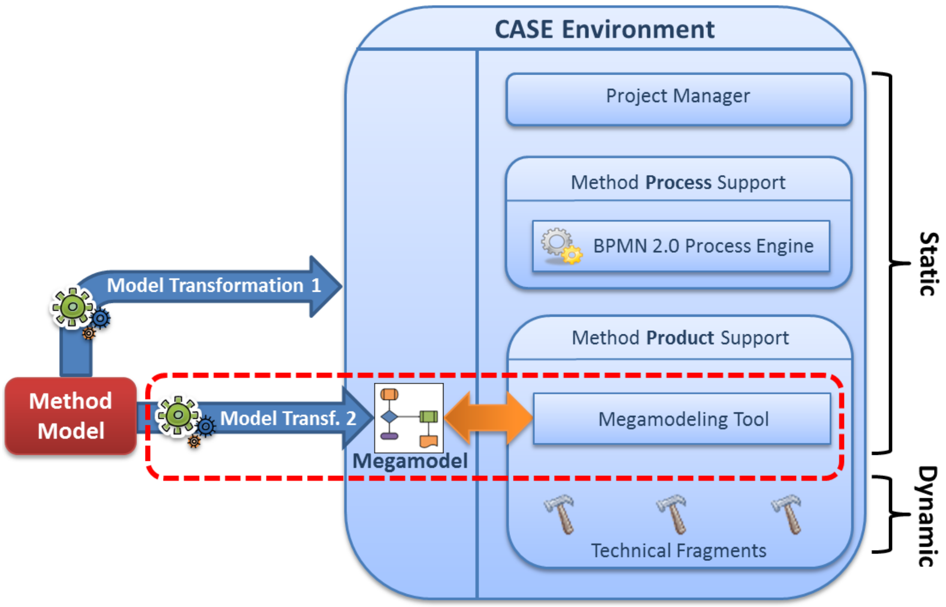


FIGURE 7.3: Megamodeling in our methodological approach

model management, and, at the same time, are adapted to the situation at hand. With the aim of providing this systematic solution, we plan to extend our methodological approach with **megamodeling** techniques [200, 201]. By making use of megamodeling, software engineers will be able to represent global entities (such as models, metamodels, and software tools) within a single (mega)model; then, this megamodel can be used to, for example, obtain visual representations of the relationships between the models by simply opening the megamodel in a graphical editor. Since megamodeling represents a global, centralized, and high-level solution for model management, we believe that it can alleviate the complexity that entails the development of CASE environments that support an intensive use of models.

Figure 7.3 graphically illustrates how we plan to extend our methodological approach with megamodeling techniques (see red rounded rectangle). Specifically, we will develop an additional model transformation that obtains a megamodel from the method that is defined by method engineers. This megamodel, which will be integrated in the software environment that is generated in the method implementation phase, will evolve during the course of the method execution. The megamodel will contain, among other data, information about the products that are produced during software development. To enable the manipulation of the megamodel, we will enhance the static part of the software environment with an additional component called "Megamodeling Tool"[6]. Thanks to this component, the environment will be able to provide enhanced functionality; for instance, the Product Explorer view will allow users to graphically display relationships between the products at different granularity levels.

7.4.4 Method Analysis and Monitoring

As Section 2.1 describes, the lifecycle of Method Engineering comprises the phases of analysis, design, implementation, execution, and monitoring. In this thesis, we address three of these phases: design, implementation, and execution. With the aim to providing complete support for Method Engineering, we will extend our methodological approach so that it also covers the analysis and monitoring phases.

The **analysis phase** of the Method Engineering lifecycle, by analogy with Software Engineering [203], encompasses the tasks that aim to determine the needs or conditions to be meet by a new method; that is, the features (e.g., design tasks or managerial activities) that are expected to be present in the method that will be defined in the next Method Engineering phases. Within the Method Engineering community, it is generally agreed that the elements of a method are determined by the characteristics of the project where the method will be applied [3, 38, 39, 134] (e.g., if the project involves the development of a system that does not require data persistence, then the method tasks that relate to database design can be omitted); for this reason, we will consider project characteristics to be central elements of the method analysis phase. Specifically, in our approach, method requirements will be specified by means of natural-language documents and the characteristics of the projects will be specified by means of feature models. These feature models will enable the automated production of situational methods, as Section 7.4.1 illustrates.

The **monitoring phase** of the Method Engineering lifecycle involves the observation of the projects' progress so that appropriate corrective actions can be taken when the software engineers' performance deviates significantly from the plan.

[6]As an example of megamodeling tool, see the Eclipse AM3 toolset [202].

Monitoring is typically performed by measuring the actual values of project planning parameters (e.g., costs, schedule, and effort), comparing actual values to the estimates in the plan, and identifying significant deviations [204]. In order to support the monitoring phase in our methodological approach, we will carry out two main extensions. First, we will define and implement a M2M transformation for mapping SPEM 2.0 models into project plans; thus, method engineers will be able to define, for example, cost and schedule baselines, which are not supported by SPEM 2.0. Second, we will add a new view to the Project Manager component; this view will show monitoring data when the role of the user is project manager. Note that the BPMN 2.0 process engine does not require any extension since the current version of the Activiti Engine stores history information about ongoing and past process instances. This information can be examined by means of the "History Service" of the engine API.

Appendix A

Comparative Analysis of SPEM 2.0 and BPMN 2.0

This appendix presents a comparative analysis between the SPEM 2.0 and BPMN 2.0 standards. The comparative analysis applies the evaluation framework that is presented by Niknafs *et al.* in [21]. This framework is organized as a number of evaluation criteria (EC), each describing a characteristic or requirement that a Process Modeling Language needs to address in order to be considered suitable for use in a Method Engineering context. These criteria are divided into two groups: method modeling criteria and process support criteria. We present these two groups in Sections A.1 and A.2, respectively. For each criterion, a short description is given first; then, the criterion is applied to SPEM 2.0 and BPMN 2.0. Finally, Section A.3 presents the conclusions of the comparative analysis.

Note that we omit in this appendix the "Coverage of Method Engineering lifecycle" criterion. The reason for this is that, in our analysis, we focus exclusively on the design and execution phases, which are covered, respectively, by the "modeling support" and "process execution" criteria.

A.1 Method Modeling Criteria

In order to be suitable for use in a Method Engineering context, a Process Modeling Language must provide adequate mechanisms for modeling software development methods. The criteria of the evaluation framework that assess the fulfillment of this requirement are the following:

EC1. Modeling support. This criterion refers to the ability of a Process Modeling Language to express the elements of a method in a clear and natural way. As we mention in Chapter 1, methods are generally composed of two main parts: *product* and *process*. To the product and process parts, some authors add the method *people* and *tools* [3, 24, 25]. In accordance with these authors, we consider that a method definition must, at least, contain the following elements: the products to be created and/or consumed, the process to be followed, the people that are involved in this process, and the software tools to be used during the method execution.

- **SPEM 2.0** provides adequate concepts for modeling method elements. The main SPEM 2.0 concepts for representing the (1) product, (2) process, (3) people and (4) tool aspects of methods are the following: (1) *Work Product Definition*; (2) *Activity*, *Task Definition*, *Work Sequence*, and *Milestone*; (3) *Role Definition*, *Qualification*, and *Role Set*; and (4) *Tool Definition*.
- **BPMN 2.0** provides obscure concepts for modeling method elements (mainly because it is more oriented towards process modeling). The main BPMN 2.0 concepts for representing the (1) product and (2) process parts of methods are, respectively, (1) *DataObject*, and (2) *Activity* (and all its subclasses: *Task*, *Subprocess*, etc.) and *Sequence Flow*. The (3) people dimension is usually modeled using the *Lane* primitive. Another possibility is to associate generic *Resources* to *Activities* by means of *Performer* elements. The (4) tool dimension can be modeled in different ways depending on the type of task. For *User Tasks* the *Rendering* element can be used as an extensible mechanism for specifying User Interface renderings. For *Script Tasks* and *Service Tasks* the attributes “script” and “operationRef” establish, respectively, the script and operation that will be invoked during the task execution.

EC2. Abstraction/Modularization. This criterion enables the evaluation of the extent to which a Process Modeling Language supports the organization and modularization of reusable method content.

- **SPEM 2.0** proposes powerful mechanisms for designing and managing maintainable, large-scale, reusable, and configurable libraries or repositories of method content. These mechanisms enable SPEM 2.0 to be used as a framework for the construction of software development knowledge bases, where reusable method content can be stored in a standardized manner. Note that this feature of SPEM 2.0 aligns with one of the most common Method Engineering approach, the assembly-based approach [9], since it facilitates the construction of method bases that store reusable method parts.
- **BPMN 2.0** processes are by definition reusable. They can be invoked by other processes by means of *Call Activities*. However, BPMN 2.0 does not define mechanisms for defining libraries or repositories of reusable process elements.

EC3. Formalism. This criterion refers to the way a Process Modeling Language represents method elements. The more formal rigor is applied, the more automated support is possible, but the language gets less understandable. The best approach is to use an intuitive graphical representation that is built upon a formal textual language.

- **SPEM 2.0** defines a visual, semi-formal UML-based language. Formal features are not considered, but seem feasible using the Object Constraint Language (OCL).
- **BPMN 2.0** defines a graphical notation and it also formalizes process execution semantics, thereby bridging the gap between process design and process implementation. In addition, BPMN 2.0 defines mappings with WS-BPEL, a formal XML-based language for the definition of executable business processes.

EC4. Simplicity. This criterion refers to the clarity, ease of use, and understandability of a Process Modeling Language. It is commonly believed that these features are related to the adoption of a graphical notation, since "pictures" are closer (as compared to text) to the cognitive part of the human brain [205].

- **SPEM 2.0** does not address simplicity, due to its complex structure. This makes the language difficult to learn.
- **BPMN 2.0** defines an intuitive graphical notation that has been especially designed for use by the people who design and manage business processes. This makes the language easy to learn.

A.2 Process Support Criteria

In order to be suitable for use in a Method Engineering context, a Process Modeling Language must not overlook the process aspects of methods. The evaluation criteria that assess the fulfillment of this requirement are the following:

EC5. Process Execution. This criterion evaluates whether the syntax of a Process Modeling Language has underlying executable semantics, allowing the processes to be executed.

- **SPEM 2.0** does not provide concepts for executing process models, but proposes two alternative ways to do so: (1) mapping processes into project plans and executing these plans using project planning tools such as Microsoft Project; or (2) mapping processes into executable languages such as BPMN 2.0 and executing the resulting processes by means of a process engine.
- **BPMN 2.0** fully formalizes process execution semantics. Thus, the process models that are compliant with BPMN 2.0 can be executed by means of process engines that provide enactment facilities such as activity orchestration, transaction management, and event/exception handling.

EC6. Process Elicitation. This criterion refers to the extent to which a Process Modeling Language supports the representation of complete, understandable, unambiguous, and well-structured processes.

- **SPEM 2.0** provides limited support to represent complex processes since it only allows method engineers to establish precedence relationships between tasks. Thus, SPEM 2.0 provides poor support to workflow patterns such as *Synchronization* and *Exclusive Choice*[1].
- **BPMN 2.0** defines powerful mechanisms to represent complete processes that support a large number of workflow patterns.

EC7. Process Evolution. This criterion evaluates whether a Process Modeling Language provides mechanisms for facilitating the resumption of process model executions after the modification of the process model (without altering previous states of artifacts, process activities, etc.).

- **SPEM 2.0** does not address process evolution.
- **BPMN 2.0** does not address process evolution.

EC8. Process Evaluation. This criterion assesses whether a Process Modeling Language supports the evaluation of process models; for instance, via the collection of process execution data and the subsequent comparison of these data with predefined process objectives.

- **SPEM 2.0** does not directly address process evaluation since it does not support the collection of execution data.
- **BPMN 2.0** does not directly address process evaluation. However, since it is an executable language, implementations can perform evaluation based on the execution data that is retrieved from the process engine.

A.3 Conclusions

This appendix illustrates by means of a comparative analysis that both SPEM 2.0 and BPMN 2.0 have advantages and limitations. This is shown in Table A.1, which summarizes the results of the analysis. A “+” symbol indicates that a Process Modeling Language fulfills a specific evaluation criterion; a “-” symbol indicates the opposite; and a “+/-” symbol indicates that a criterion is partially fulfilled.

[1] http://www.workflowpatterns.com/

TABLE A.1: Summary of the analysis of SPEM 2.0 and BPMN 2.0

	Method Modeling				**Process Support**			
	EC1	**EC2**	**EC3**	**EC4**	**EC5**	**EC6**	**EC7**	**EC8**
SPEM 2.0	+	+	+/-	-	-	+/-	-	-
BPMN 2.0	+/-	+/-	+	+	+	+	-	+/-

TABLE A.2: Evaluation criteria that are covered by this thesis

	Method Modeling				**Process Support**			
	EC1	**EC2**	**EC3**	**EC4**	**EC5**	**EC6**	**EC7**	**EC8**
SPEM 2.0	✓	✓						
BPMN 2.0			✓	✓	✓	✓		

In this thesis, our goal is to obtain benefits from both SPEM 2.0 and BPMN 2.0, and, for this reason, we define a DSL (see Chapter 4) that combines the two languages. By combining the two languages, we cover six of the eight evaluation criteria that are described in this appendix. These six criteria are supported by either SPEM 2.0 or BPMN 2.0, as shown in Table A.2. Specifically, SPEM 2.0 supports the first two criteria because it provides suitable concepts for method modeling (EC1) – e.g., task, role, and work product – as well as mechanisms for defining repositories of reusable method content (EC2). On the other hand, BPMN 2.0 strikes a good balance between formalism (EC3) and simplicity (EC4) – that is, it provides an intuitive graphical notation, while, at the same time, applies enough formal rigor for enabling execution – while it also fully formalizes process execution semantics (EC5) and provides suitable workflow-related primitives (EC6). Finally, neither SPEM 2.0 nor BPMN 2.0 support process evolution (EC7) and process evaluation (EC8), and, therefore, these criteria are not covered in this thesis.

In a nutshell, based on the results of the comparative analysis that is presented in this appendix, we define a DSL that integrates concepts from both SPEM 2.0 and BPMN 2.0. While the concepts of SPEM 2.0 enable the definition of complete methods, the concepts of BPMN 2.0 allow method engineers to enhance the method process part (by means of primitives that are not supported in SPEM 2.0, such as gateways or different types of tasks). Additionally, we leverage BPMN 2.0 by implementing a M2M transformation that obtains executable representations of the method models that are defined using our DSL.

Appendix B

A Case Study: the gvMétrica Method

This appendix introduces a case study that was carried out in the context of the Valencian Regional Ministry of Infrastructure, Territory, and Environment (also known as CITMA). Within this public entity, we put into practice the methodological approach that is presented in Chapter 4: we used MOSKitt4ME to design gvMétrica[1] and also to generate a software environment that supports part of the method. Overall, the application of MOSKitt4ME in an industrial context was successful; this is a good indicator of the feasibility of the model-driven Method Engineering approach that is presented in this thesis.

The remainder of the present appendix is structured as follows. First, Section B.1 introduces gvMétrica. Due to the large size of gvMétrica, this section does not provide a thorough description of the method; rather, it gives a general overview first, and, then, it details a small excerpt: the phase of information systems design. We selected this excerpt of gvMétrica for our case study for two main reasons. First, it represents a simple, understandable, and realistic scenario that includes enough elements for the complete application of our Method Engineering approach. Second, it is the part of gvMétrica that is best supported by the software environment that was obtained during the case study. The application

[1] http://www.gvpontis.gva.es/cast/proyectos-integra/

of the three phases that comprise our Method Engineering approach – design, implementation, and execution – is described in Section B.2. Finally, Section B.3 outlines some conclusions about the case study.

B.1 The gvMétrica Method

GvMétrica is the result of the adaptation (to the needs of the CITMA) of a software development method that is called Métrica III[2]. The Métrica III method was defined by the Spanish government in 2001 and its main goal is to provide organizations with an instrument that systematizes the activities that support the lifecycle of software applications. Métrica III is not only designed to enable its adaptation to the needs of particular projects or organizations, but it also explicitly recommends to carry out this adaptation. For this reason, the CITMA adapted Métrica III during the period between 2003 and 2006. The result was gvMétrica.

The specification of gvMétrica by practitioners from the CITMA involved several activities, which were documented as part of the gvPONTIS project[3]. Examples of these activities are the definition of templates for reporting deliverables and the selection of the processes of Métrica III that were needed for the CITMA. The result of the process selection included three processes that comprise the general structure of gvMétrica: planning of information systems (PSI), development of information systems (DSI), and maintenance of information systems (MSI). Of these three main processes, the one that falls within the scope of this thesis is the DSI process since it deals with the development of software systems. The overall structure of the DSI process is composed of five major phases: study of the feasibility of the system (EVS), analysis of the information system (CASI), design of the information system (CDSI), construction of the information system (CCSI), and deployment and acceptance of the system (CIAS). In the following subsection, we detail the CDSI phase since it represents the test object of the case study that is described in this appendix.

[2] http://administracionelectronica.gob.es/pae_Home/pae_Documentacion.html
[3] http://www.gvpontis.gva.es/cast/inicio-gvpontis/

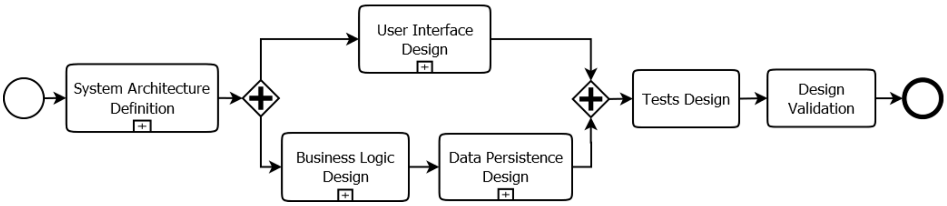
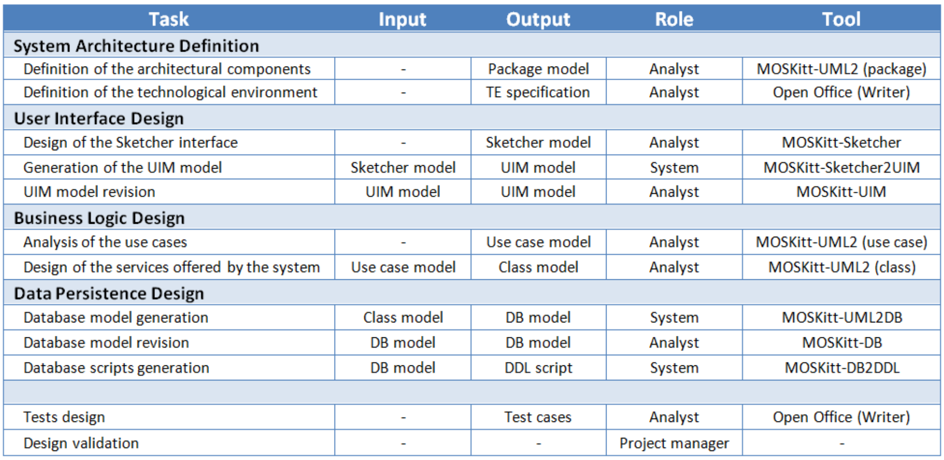


Figure B.1: Overall process of the CDSI phase of gvMétrica



Task	Input	Output	Role	Tool
System Architecture Definition				
Definition of the architectural components	-	Package model	Analyst	MOSKitt-UML2 (package)
Definition of the technological environment	-	TE specification	Analyst	Open Office (Writer)
User Interface Design				
Design of the Sketcher interface	-	Sketcher model	Analyst	MOSKitt-Sketcher
Generation of the UIM model	Sketcher model	UIM model	System	MOSKitt-Sketcher2UIM
UIM model revision	UIM model	UIM model	Analyst	MOSKitt-UIM
Business Logic Design				
Analysis of the use cases	-	Use case model	Analyst	MOSKitt-UML2 (use case)
Design of the services offered by the system	Use case model	Class model	Analyst	MOSKitt-UML2 (class)
Data Persistence Design				
Database model generation	Class model	DB model	System	MOSKitt-UML2DB
Database model revision	DB model	DB model	Analyst	MOSKitt-DB
Database scripts generation	DB model	DDL script	System	MOSKitt-DB2DDL
Tests design	-	Test cases	Analyst	Open Office (Writer)
Design validation	-	-	Project manager	-

Table B.1: Elements of the CDSI phase of gvMétrica

B.1.1 Design of Information Systems in gvMétrica

The CDSI phase of gvMétrica involves a set of processes and tasks that are carried out according to the workflow that is graphically depicted in Figure B.1. As this figure shows, the CDSI phase comprises four processes (i.e., *System Architecture Definition*, *User Interface Design*, *Business Logic Design*, and *Data Persistence Design*) and two tasks (i.e., *Tests Design* and *Design Validation*). All of these processes and tasks are further detailed in Table B.1. As one can observe in this table (in the leftmost column), the processes of the CDSI phase are decomposed into various tasks, which, in turn, reference their input and output products, the roles that are in charge of the tasks performance, and the tools that enable the tasks execution. Below, we describe the elements that are involved in the CDSI phase.

System Architecture Definition. During this process, analysts perform two tasks that can be performed in parallel:

- Definition of the components that comprise the software architecture of the system. To do this, analysts must use the MOSKitt-UML2 tool to create a UML 2.0 package model.
- Specification of the technological environment of the project (i.e., the required software tools) using a set of predefined textual templates. These templates must be filled by means of the Writer application that is provided as part of the Open Office tool suite.

Once the *System Architecture Definition* process is completed, two different processes start in parallel: *User Interface Design* and *Business Logic Design*.

User Interface Design. In this process, software engineers define the graphical user interface of the system. To this end, they perform three sequential tasks:

- Definition at a high level of abstraction of the graphical components (e.g., windows, menus, and buttons) that comprise the user interface of the system. To do this, analysts must use a graphical editor: the MOSKitt-Sketcher tool. The output of this task is, therefore, a model that conforms to the MOSKitt-Sketcher metamodel. This output model does not specify, for example, the behavior of the graphical components; it only contains graphical information, such as position and size.
- Generation of a lower-level model of the graphical user interface. This task is automated by a M2M transformation: MOSKitt-Sketcher2UIM. This transformation takes the sketcher model as input and obtains a model that conforms to the MOSKitt-UIM metamodel. This metamodel provides more expresiveness than the MOSKitt-Sketcher metamodel.
- Analysts revise the generated UIM model using a graphical editor: the MOSKitt-UIM tool. The main goal of this task is to enhance the definition of the user interface; for instance, by specifying the behavior of its graphical components.

Business Logic Design. During this process, analysts perform two sequential tasks:

- Specification of the use cases of the system. To do this, analysts must use the MOSKitt-UML2 tool to create a UML 2.0 use case diagram.
- Specification of the services that the system will offer to other systems. To do this, analysts must use the MOSKitt-UML2 tool; in this case, analysts must build a UML 2.0 class diagram.

Once the *Business Logic Design* process is completed, *Data Persistence Design* can start.

Data Persistence Design. In this process, three tasks are performed sequentially:

- Generation of a database schema model from the UML 2.0 class model. The database model is automatically obtained by means of a M2M transformation: MOSKitt-UML2DB.
- The analyst manually revises the database schema model using a graphical editor: the MOSKitt-DB tool.
- Automatic generation of the DDL code that implements the database schema. The DDL code is generated by means of a M2T transformation: MOSKitt-DB2DDL.

Once all of the system artifacts (i.e., the specification of the graphical user interface, the UML 2.0 models, the database schema model, the DDL code, etc.) have been obtained, the next task to be performed is called *Tests Design*. In this task, analysts define a set of test cases for the software system. To do this, they must use the Writer application that is provided as part of the Open Office tool suite. Finally, the last task to be performed is *Design Validation*. In this task, the project leader validates all the work performed in the CDSI phase.

B.2 Applying the Methodological Approach

This section describes how we applied our methodological approach to design, implement, and execute the CDSI phase of gvMétrica. We divide this section into three subsections (B.2.1, B.2.2, and B.2.3), each of which focuses on a specific phase of the approach.

B.2.1 Method Design

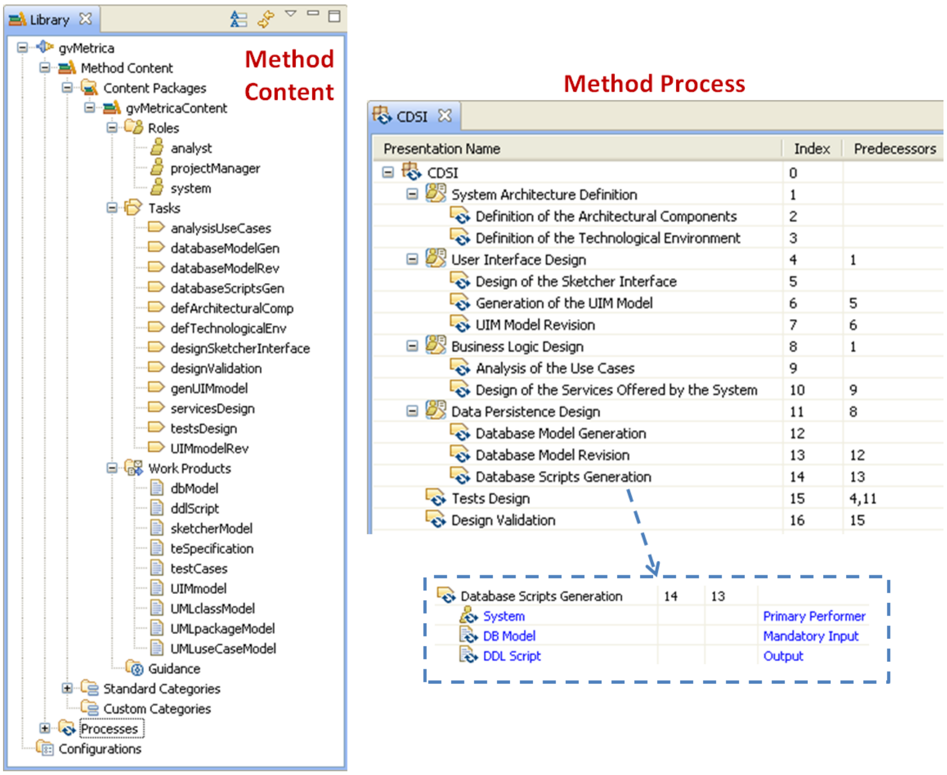
As Section 4.2.3 describes, our approach for method design is composed of three major steps: *method definition*, *method configuration*, and *executable process generation*. The application of these three steps to the case study is presented in Sections B.2.1.1, B.2.1.2, and B.2.1.3, respectively.

B.2.1.1 Method Definition

The definition of the CDSI phase of gvMétrica was performed by means of the EPF Composer editor, which is integrated in MOSKitt4ME (see Section 5.3.1.1). The result of the method definition is shown in Figure B.2. As the figure shows on the left hand side, the method content, which was defined by means of the Library view, comprises three roles, twelve tasks, and nine work products, all of which were extracted from the data that is presented in Table B.1. For instance, the task "databaseScriptsGen" corresponds to the task "Database Scripts Generation". Note that the tools of the method are not displayed in Figure B.2. This is because the method tools are defined in the next step of our approach (see Section B.2.1.2).

On the other hand, the right side of Figure B.2 shows the process part of the case study after being defined by means of the process editor of the EPF Composer. As one can observe in the figure, the activities of the process represent processes from Table B.1 (e.g., "Business Logic Design") and the task descriptors represent tasks (e.g., "Design Validation"). These activities and task descriptors form a work breakdown structure where precedence relationships are established by means of the "predecessors" column (e.g., the predecessor of the activity "User Interface Design" is the activity "System Architecture Definition"). Note that the process editor also allows the user to examine the roles, inputs, and outputs of the task descriptors, as illustrated in the bottom-right part of Figure B.2.

It is important to highlight that, during the *Method Definition* step of our Method Engineering approach, two important limitations of MOSKitt4ME came to light.



- gvMetrica
  - Method Content
    - Content Packages
      - gvMetricaContent
        - Roles
          - analyst
          - projectManager
          - system
        - Tasks
          - analysisUseCases
          - databaseModelGen
          - databaseModelRev
          - databaseScriptsGen
          - defArchitecturalComp
          - defTechnologicalEnv
          - designSketcherInterface
          - designValidation
          - genUIMmodel
          - servicesDesign
          - testsDesign
          - UIMmodelRev
        - Work Products
          - dbModel
          - ddlScript
          - sketcherModel
          - teSpecification
          - testCases
          - UIMmodel
          - UMLclassModel
          - UMLpackageModel
          - UMLuseCaseModel
        - Guidance
    - Standard Categories
    - Custom Categories
  - Processes
- Configurations

Presentation Name	Index	Predecessors
CDSI	0	
System Architecture Definition	1	
Definition of the Architectural Components	2	
Definition of the Technological Environment	3	
User Interface Design	4	1
Design of the Sketcher Interface	5	
Generation of the UIM Model	6	5
UIM Model Revision	7	6
Business Logic Design	8	1
Analysis of the Use Cases	9	
Design of the Services Offered by the System	10	9
Data Persistence Design	11	8
Database Model Generation	12	
Database Model Revision	13	12
Database Scripts Generation	14	13
Tests Design	15	4,11
Design Validation	16	15

Database Scripts Generation	14	13	
System			Primary Performer
DB Model			Mandatory Input
DDL Script			Output

FIGURE B.2: Definition in EPF Composer of the CDSI phase of gvMétrica

First, MOSKitt4ME does not allow users to specify method variability, a capability that is desirable in the context of the CITMA. While MOSKitt4ME supports the definition and reuse of method fragments, it does not enable the association of these method fragments with project characteristics. This association would allow MOSKitt4ME to automatically obtain project-specific methods directly from the characteristics of the projects. Second, more expresiveness – for instance, the expresiveness that is provided by Responsibility Assignment Matrices (RAM) – is needed to specify the roles that are defined in gvMétrica. In its current version, MOSKitt4ME only supports the specification of primary and additional performers for the tasks.

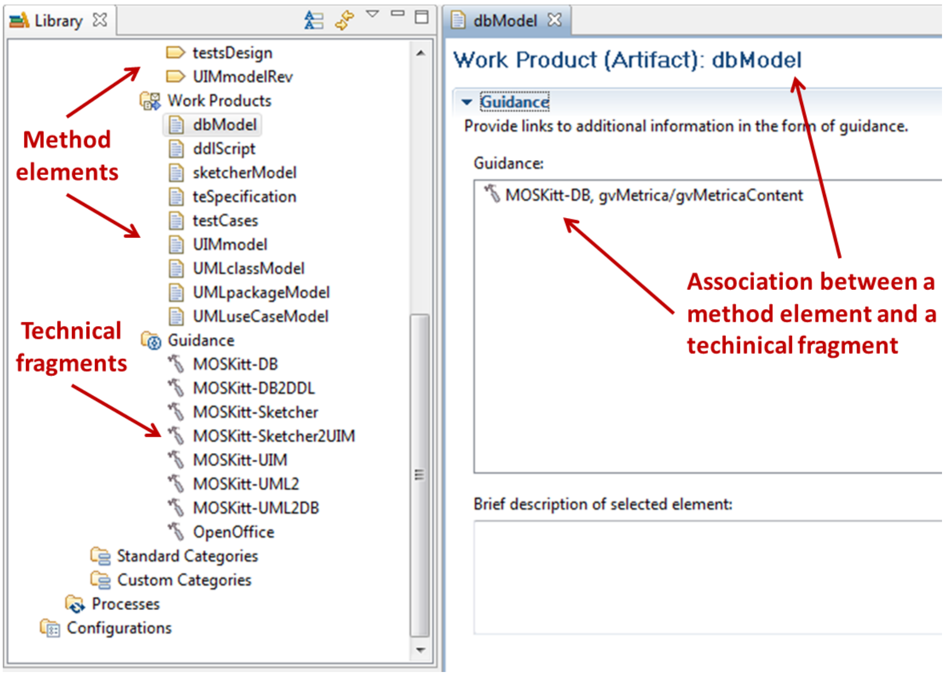
Table B.2: Associations between technical fragments and method elements

Technical fragment	**Supported Elements**
MOSKitt-UML2.ras	Package model Use case model Class model
OpenOffice.ras	TE specification Test cases
MOSKitt-Sketcher.ras	Sketcher model
MOSKitt-Sketcher2UIM.ras	Generation of the UIM model
MOSKitt-UIM.ras	UIM model
MOSKitt-UML2DB.ras	Database model generation
MOSKitt-DB.ras	DB model
MOSKitt-DB2DDL.ras	Database scripts generation

B.2.1.2 Method Configuration

This step was performed by means of the repository client of MOSKitt4ME (see Section 5.3.1.2), which enabled the specification of the method tools by means of the association of method elements with technical fragments. The associations that resulted from the *Method Configuration* step are shown in Table B.2. As the table shows, eight technical fragments were required to support the CDSI phase of gvMétrica[4]. Seven of these fragments encapsulate MOSKitt components, while the "OpenOffice.ras" fragment represents an external tool. All of these technical fragments were associated either to method products or method tasks. As an example of association between a technical fragment and a method element, let us consider the task "Database model generation". The execution of this task obtains a database model from a UML 2.0 class model. To specify this behavior, we associated the task with a technical fragment that contains the Eclipse plug-ins that implement the UML2DB transformation that is provided by MOSKitt. Thus, we specified that this model transformation will be launched when the "Database model generation" task is active during the method execution. On the other hand, the output product of this task (i.e., the product "DB model") was associated to the technical fragment "MOSKitt-DB"; thus, we specified that the

[4]Even though it is not shown in the table, additional fragments were required to satisfy the software dependencies of these eight fragments; nonetheless, the resolution of dependencies was transparent to the users since it is automatically performed by MOSKitt4ME.



Provide links to additional information in the form of guidance.

Figure B.3: Technical fragments in the EPF Composer

graphical editor that is contained in the technical fragment will be used during the method execution for the manipulation of the database model.

Figure B.3 shows a screenshot of MOSKitt4ME after we performed the *Method Configuration* step of our approach. As the figure shows, the Library view of the EPF Composer shows (under the "Guidance" folder) the eight technical fragments that were retrieved from the repository and associated to method elements. The right part of the figure illustrates how the user can examine the associations between technical fragments and method elements. Specifically, these associations can be examined in the "Guidance" tab of the editor that is opened when the user selects a task or a product in the Library view.

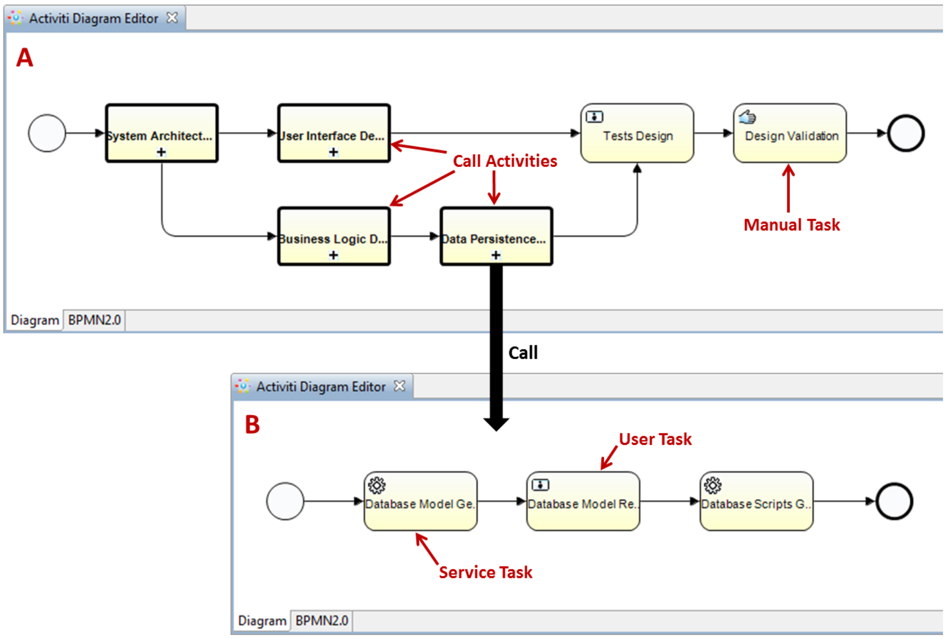


FIGURE B.4: Example of generated BPMN 2.0 processes

B.2.1.3 Executable Process Generation

This step was performed automatically by means of the SPEM2BPMN transformation that is provided by MOSKitt4ME (see Section 5.3.1.1). As an example of the result obtained in the *Executable Process Generation* step, Figure B.4 shows two of the BPMN 2.0 processes that were generated after applying the SPEM2BPMN transformation to the case study. These processes are represented in terms of the Activiti Designer, the Eclipse-based graphical editor that is integrated in MOSKitt4ME to support BPMN 2.0.

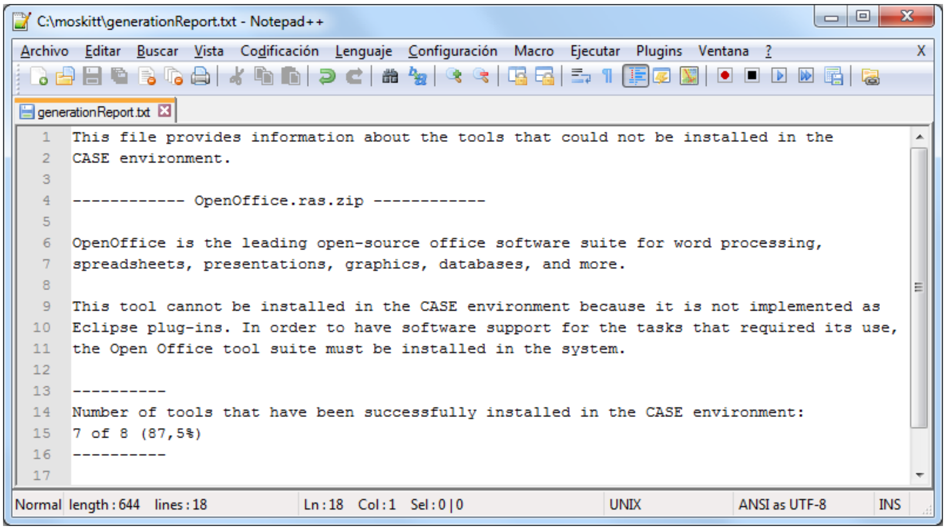
To illustrate how the BPMN 2.0 processes that are shown in Figure B.4 were generated, we describe below some mappings between elements of the method model (which was defined in the previous two steps) and BPMN 2.0 elements.

- The *Process* "CDSI" of the method model was mapped into a BPMN 2.0 *Process* (diagram "A" in Figure B.4).

- The *Activity* "Data Persistence Design" of the method model was mapped into a *Call Activity* and a BPMN 2.0 *Process* (diagram "B" in Figure B.4).
- The *Task* "Database Model Generation" of the method model was mapped into a BPMN 2.0 *Service Task* since this task is considered automatic. This is because the task has a M2M transformation associated to it as a technical fragment (i.e., "MOSKitt-UML2DB").
- The *Task* "Database Model Revision" of the method model was mapped into a BPMN 2.0 *User Task* since this task must be performed manually by means of a software application. This is because the task has an output product with a technical fragment associated to it. This product is "DB model" (which is not shown in Figure B.4) and it was associated to the technical fragment "MOSKitt-DB".
- The *Task* "Design Validation" of the method model was mapped into a BPMN 2.0 *Manual Task* since this task must be performed manually without using any software application. This is because none of the technical fragments was associated to the task (nor to its output products).
- The *Sequences* of the method model were mapped into the *Sequence Flows* that connect the BPMN 2.0 elements in both diagrams.

B.2.2 Method Implementation

The method implementation phase of our approach is automatic; therefore, we did not have to perform any work at this point. The result of the method implementation phase was (1) a software environment that provides support to the method of the case study and also (2) a generation report. The generation report is shown in Figure B.5. As the figure shows, seven of the eight technical fragments were successfully installed in the software environment. These seven technical fragments correspond to the components of MOSKitt (e.g., the MOSKitt-UML2DB transformation and the MOSKitt-UIM editor); these components could be successfully installed since they are implemented as Eclipse plug-ins. In contrast, the technical fragment "OpenOffice.ras" could not be installed since it is implemented in a different technology. In this case, the generation report indicates that software


C:\moskitt\generationReport.txt - Notepad++
Archivo Editar Buscar Vista Codificación Lenguaje Configuración Macro Ejecutar Plugins Ventana ?
generationReport.txt
Normal length : 644 lines : 18 Ln : 18 Col : 1 Sel : 0 | 0 UNIX ANSI as UTF-8 INS

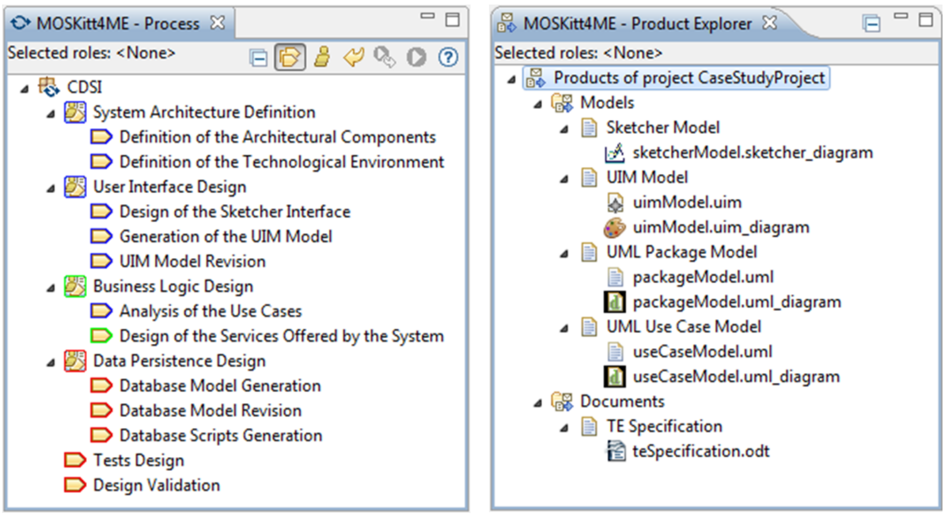
```
This file provides information about the tools that could not be installed in the
CASE environment.

------------ OpenOffice.ras.zip ------------

OpenOffice is the leading open-source office software suite for word processing,
spreadsheets, presentations, graphics, databases, and more.

This tool cannot be installed in the CASE environment because it is not implemented as
Eclipse plug-ins. In order to have software support for the tasks that required its use,
the Open Office tool suite must be installed in the system.

----------
Number of tools that have been successfully installed in the CASE environment:
7 of 8 (87,5%)
----------
```

FIGURE B.5: Generation report obtained in the case study

engineers must install Open Office so that this tool is available during the phase of method execution.

B.2.3 Method Execution

In order to illustrate the method execution phase of the case study, we created an example project by means of the Project Explorer view of the software environment that was generated in the method implementation phase. When this new project is selected in the Project Explorer view, the Activiti Engine starts a new process instance and the Process and Product Explorer views are updated accordingly. At this point, the Process view shows the initial state of the process and the Product Explorer view is empty. Now, let us consider that the following six tasks have been executed: definition of the architectural components, definition of the technological environment, design of the sketcher interface, generation of the UIM model, UIM model revision, and analysis of the use cases. Thus, the next executable task is "Design of the Services Offered by the System". This state of the process is shown in Figure B.6.



MOSKitt4ME - Process

Selected roles: <None>

- CDSI
  - System Architecture Definition
    - Definition of the Architectural Components
    - Definition of the Technological Environment
  - User Interface Design
    - Design of the Sketcher Interface
    - Generation of the UIM Model
    - UIM Model Revision
  - Business Logic Design
    - Analysis of the Use Cases
    - Design of the Services Offered by the System
  - Data Persistence Design
    - Database Model Generation
    - Database Model Revision
    - Database Scripts Generation
  - Tests Design
  - Design Validation

MOSKitt4ME - Product Explorer

Selected roles: <None>

- Products of project CaseStudyProject
  - Models
    - Sketcher Model
      - sketcherModel.sketcher_diagram
    - UIM Model
      - uimModel.uim
      - uimModel.uim_diagram
    - UML Package Model
      - packageModel.uml
      - packageModel.uml_diagram
    - UML Use Case Model
      - useCaseModel.uml
      - useCaseModel.uml_diagram
  - Documents
    - TE Specification
      - teSpecification.odt

FIGURE B.6: Process and Product Explorer views

The Process view (left part of Figure B.6) shows the tasks and activities in different colors depending on whether they have already been executed (blue), are executable (green), or are not executable (red) in the current state of the process. Note that, even though the Process view shows the process in terms of SPEM 2.0, the process instance corresponds to an instance of a BPMN 2.0 process. This is possible because there is a one-to-one correspondence between SPEM 2.0 tasks and BPMN 2.0 tasks.

On the other hand, the Product Explorer view (right part of Figure B.6) depicts the artifacts that have been produced during the method execution. In this case, the Product Explorer is showing the output products of the six tasks that have been executed.

Now, let us consider that the user wants to proceed with the method execution. To do this, the user must select the task "Design of the Services Offered by the System" since it is the only task that is displayed in green. When this task is executed (along with the remaining five tasks), the Activiti Engine deletes the process instance, and, then, the project can be considered to be concluded.

B.3 Conclusions

This appendix presents a case study that exemplifies the model-driven Method Engineering approach that is presented in this thesis. To this end, the appendix applies the approach to an example method, which represents an excerpt of the software development method that was defined in the CITMA: gvMétrica. The application of the approach to an industrial method allowed us to identify some limitations of MOSKitt4ME (such as the lack of support for variability and the limited expresiveness that MOSKitt4ME provides for role definition); nonetheless, it also allows us to be optimistic since MOSKitt4ME successfully supported the design, implementation, and execution of the case study.

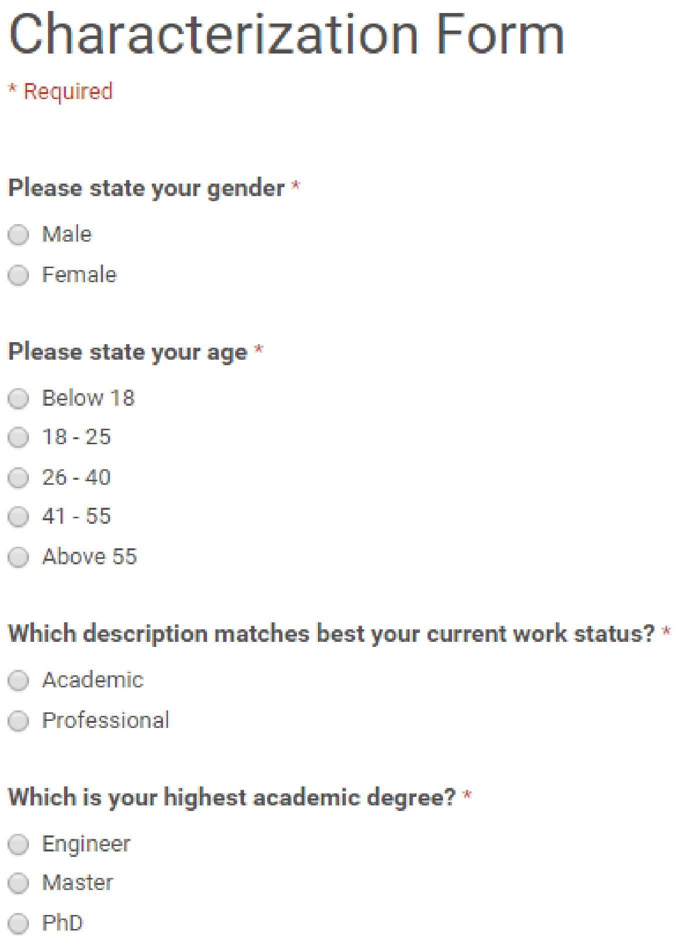
Appendix C

Supplementary Material on the Evaluation Study

This appendix includes material that was used during the evaluation study that is presented in Chapter 6. First, the appendix presents several instruments that were employed during the execution phase of the study. These instruments are the characterization form, the user acceptance form, and the interview questions, which are presented in Sections C.1, C.2, and C.3, respectively. Then, Section C.4 details the statistical tests that were performed during data analysis. Finally, Section C.5 presents the coding scheme that was developed during the construction of the Think Aloud protocols.

C.1 Characterization Form

This section presents the characterization form. As Section 6.3.8 describes, the characterization form is divided in two parts. The first part requests demographic data, such as gender, age, and work status. This part of the form is shown in Figure C.1. On the other hand, the second part includes twelve multiple-choice questions that request the experience level of the subjects regarding the topics that



Characterization Form

* Required

Please state your gender *

- ○ Male
- ○ Female

Please state your age *

- ○ Below 18
- ○ 18 - 25
- ○ 26 - 40
- ○ 41 - 55
- ○ Above 55

Which description matches best your current work status? *

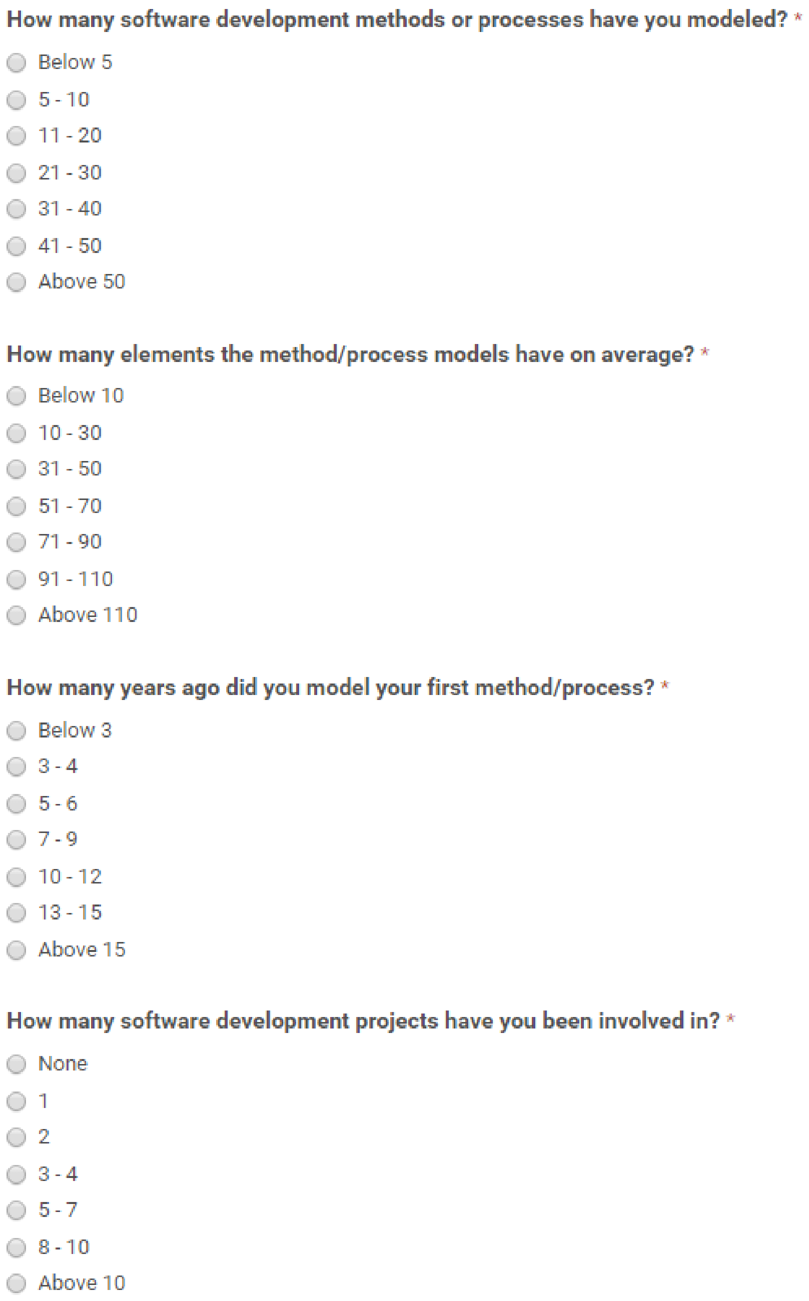
- ○ Academic
- ○ Professional

Which is your highest academic degree? *

- ○ Engineer
- ○ Master
- ○ PhD

FIGURE C.1: Characterization form: Demographic data

are covered by the study (e.g., Eclipse, SPEM 2.0, and Model-Driven Engineering). This part of the form is shown in Figures C.2 and C.3.



How many software development methods or processes have you modeled? *

○ Below 5
○ 5 - 10
○ 11 - 20
○ 21 - 30
○ 31 - 40
○ 41 - 50
○ Above 50

How many elements the method/process models have on average? *

○ Below 10
○ 10 - 30
○ 31 - 50
○ 51 - 70
○ 71 - 90
○ 91 - 110
○ Above 110

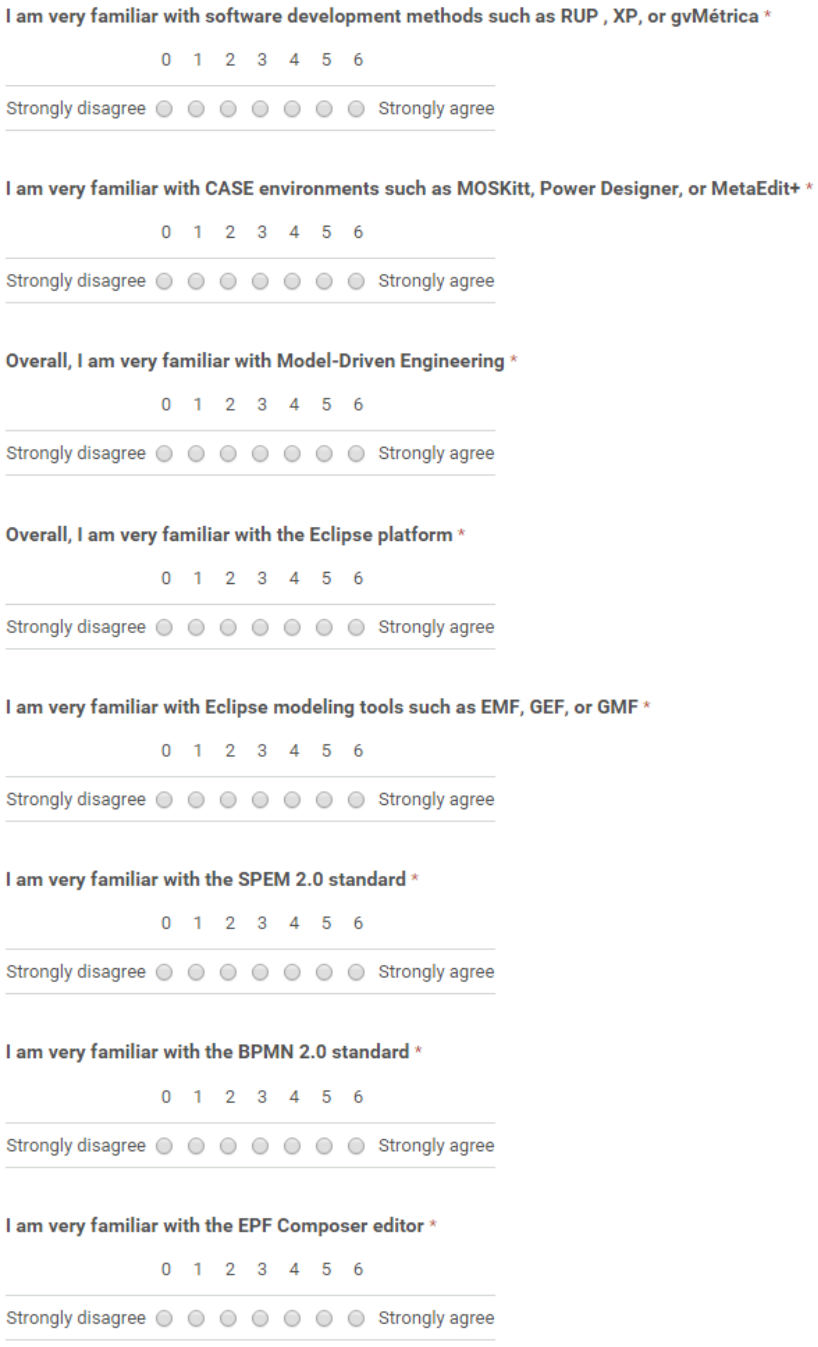
How many years ago did you model your first method/process? *

○ Below 3
○ 3 - 4
○ 5 - 6
○ 7 - 9
○ 10 - 12
○ 13 - 15
○ Above 15

How many software development projects have you been involved in? *

○ None
○ 1
○ 2
○ 3 - 4
○ 5 - 7
○ 8 - 10
○ Above 10

FIGURE C.2: Characterization form: Experience (1)



I am very familiar with software development methods such as RUP , XP, or gvMétrica *

	0	1	2	3	4	5	6	
Strongly disagree	○	○	○	○	○	○	○	Strongly agree

I am very familiar with CASE environments such as MOSKitt, Power Designer, or MetaEdit+ *

	0	1	2	3	4	5	6	
Strongly disagree	○	○	○	○	○	○	○	Strongly agree

Overall, I am very familiar with Model-Driven Engineering *

	0	1	2	3	4	5	6	
Strongly disagree	○	○	○	○	○	○	○	Strongly agree

Overall, I am very familiar with the Eclipse platform *

	0	1	2	3	4	5	6	
Strongly disagree	○	○	○	○	○	○	○	Strongly agree

I am very familiar with Eclipse modeling tools such as EMF, GEF, or GMF *

	0	1	2	3	4	5	6	
Strongly disagree	○	○	○	○	○	○	○	Strongly agree

I am very familiar with the SPEM 2.0 standard *

	0	1	2	3	4	5	6	
Strongly disagree	○	○	○	○	○	○	○	Strongly agree

I am very familiar with the BPMN 2.0 standard *

	0	1	2	3	4	5	6	
Strongly disagree	○	○	○	○	○	○	○	Strongly agree

I am very familiar with the EPF Composer editor *

	0	1	2	3	4	5	6	
Strongly disagree	○	○	○	○	○	○	○	Strongly agree

FIGURE C.3: Characterization form: Experience (2)

C.2 User Acceptance Form

This section presents the user acceptance form. As Section 6.3.8 describes, we developed the user acceptance form following the Technology Acceptance Model (TAM), which suggests measuring perceived usefulness and perceived ease of use by means of two scales of six 7-point Likert items, ranging from “strongly disagree” (0) to “neutral” (3) to “strongly agree” (6). The first of these two scales, which evaluates perceived usefulness, is graphically depicted in Figure C.4. The second scale, which evaluates perceived ease of use, is graphically depicted in Figure C.5.

C.3 Interview Questions

Performance

1. What do you think about your performance when you carried out the tasks of the study without using MOSKitt4ME? (e.g., do you think you were effective?)

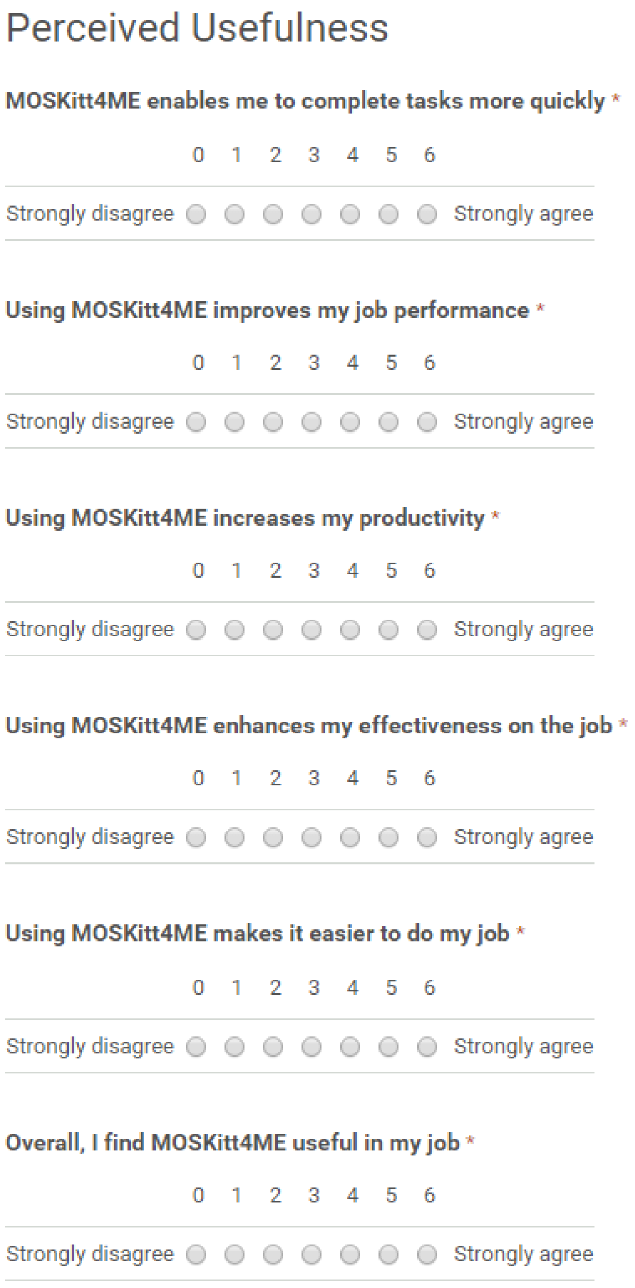
2. What do you think about your performance when you carried out the tasks of the study using MOSKitt4ME? (e.g., do you think you were effective?)

Functionality

1. What do you think about the role that MOSKitt4ME plays in automating the CASE environment construction?

2. Do you find useful the assistance that is provided by MOSKitt4ME during the method execution? Why (not)?

3. What do you think it is more useful? The textual definition of the method or the method model? Why?



Perceived Usefulness

MOSKitt4ME enables me to complete tasks more quickly *

	0	1	2	3	4	5	6	
Strongly disagree	○	○	○	○	○	○	○	Strongly agree

Using MOSKitt4ME improves my job performance *

	0	1	2	3	4	5	6	
Strongly disagree	○	○	○	○	○	○	○	Strongly agree

Using MOSKitt4ME increases my productivity *

	0	1	2	3	4	5	6	
Strongly disagree	○	○	○	○	○	○	○	Strongly agree

Using MOSKitt4ME enhances my effectiveness on the job *

	0	1	2	3	4	5	6	
Strongly disagree	○	○	○	○	○	○	○	Strongly agree

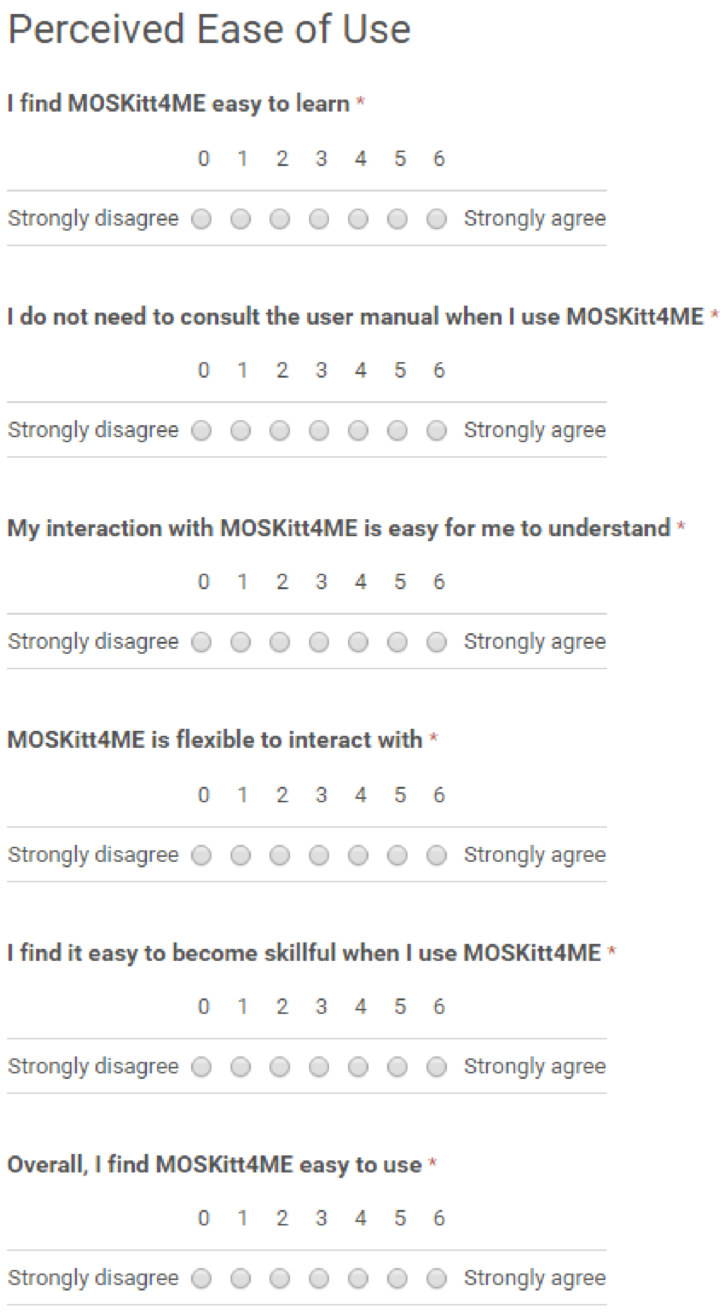
Using MOSKitt4ME makes it easier to do my job *

	0	1	2	3	4	5	6	
Strongly disagree	○	○	○	○	○	○	○	Strongly agree

Overall, I find MOSKitt4ME useful in my job *

	0	1	2	3	4	5	6	
Strongly disagree	○	○	○	○	○	○	○	Strongly agree

FIGURE C.4: User acceptance form: Perceived usefulness



Perceived Ease of Use

I find MOSKitt4ME easy to learn *

	0	1	2	3	4	5	6	
Strongly disagree	○	○	○	○	○	○	○	Strongly agree

I do not need to consult the user manual when I use MOSKitt4ME *

	0	1	2	3	4	5	6	
Strongly disagree	○	○	○	○	○	○	○	Strongly agree

My interaction with MOSKitt4ME is easy for me to understand *

	0	1	2	3	4	5	6	
Strongly disagree	○	○	○	○	○	○	○	Strongly agree

MOSKitt4ME is flexible to interact with *

	0	1	2	3	4	5	6	
Strongly disagree	○	○	○	○	○	○	○	Strongly agree

I find it easy to become skillful when I use MOSKitt4ME *

	0	1	2	3	4	5	6	
Strongly disagree	○	○	○	○	○	○	○	Strongly agree

Overall, I find MOSKitt4ME easy to use *

	0	1	2	3	4	5	6	
Strongly disagree	○	○	○	○	○	○	○	Strongly agree

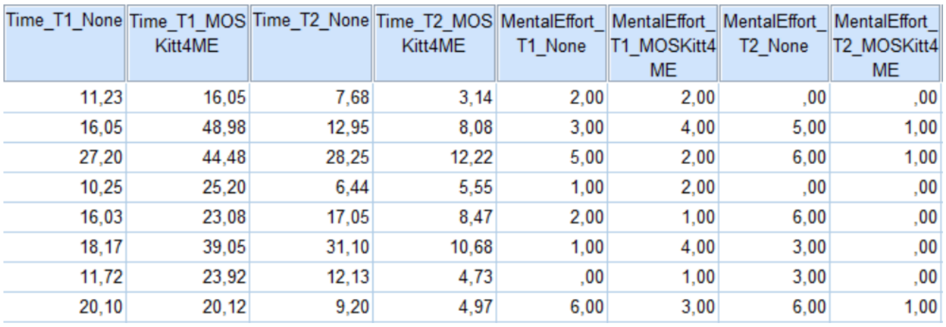
Figure C.5: User acceptance form: Perceived ease of use

C.4 Statistical Tests

This section describes the statistical tests that were carried out in the evaluation study of MOSKitt4ME. These tests, which were performed by means of the *IBM SPSS Statistics 2.0*, involve the following eight variables:

- **Time_T1_None:** the time (in minutes) that is invested by the subjects in the method design/implementation task applying Treatment 1.
- **Time_T1_MOSKitt4ME:** the time (in minutes) that is invested by the subjects in the method design/implementation task applying Treatment 2.
- **Time_T2_None:** the time (in minutes) that is invested by the subjects in the method execution task applying Treatment 1.
- **Time_T2_MOSKitt4ME:** the time (in minutes) that is invested by the subjects in the method execution task applying Treatment 2.
- **MentalEffort_T1_None:** the mental effort (0–6) that is invested by the subjects in the method design/implementation task applying Treatment 1.
- **MentalEffort_T1_MOSKitt4ME:** the mental effort (0–6) that is invested by the subjects in the method design/implementation task applying Treatment 2.
- **MentalEffort_T2_None:** the mental effort (0–6) that is invested by the subjects in the method execution task applying Treatment 1.
- **MentalEffort_T2_MOSKitt4ME:** the mental effort (0–6) that is invested by the subjects in the method execution task applying Treatment 2.

The data that was collected in the study for each of the above variables is shown in Table C.1. Each column of the table corresponds to one of the eight variables, while each row represents one subject of the study. To analyze the gathered data, we performed statistical tests that aimed to assess the differences between the results obtained in Treatment 1 and Treatment 2. To achieve this goal, two types of tests exist: parametric and non-parametric. Since we have paired samples, the parametric test that is suitable for our study is the paired t-test; however,



Time_T1_None	Time_T1_MOSKitt4ME	Time_T2_None	Time_T2_MOSKitt4ME	MentalEffort_T1_None	MentalEffort_T1_MOSKitt4ME	MentalEffort_T2_None	MentalEffort_T2_MOSKitt4ME
11,23	16,05	7,68	3,14	2,00	2,00	,00	,00
16,05	48,98	12,95	8,08	3,00	4,00	5,00	1,00
27,20	44,48	28,25	12,22	5,00	2,00	6,00	1,00
10,25	25,20	6,44	5,55	1,00	2,00	,00	,00
16,03	23,08	17,05	8,47	2,00	1,00	6,00	,00
18,17	39,05	31,10	10,68	1,00	4,00	3,00	,00
11,72	23,92	12,13	4,73	,00	1,00	3,00	,00
20,10	20,12	9,20	4,97	6,00	3,00	6,00	1,00

Table C.1: Data obtained for task completion time and mental effort

this technique requires the data to be normally distributed and the "MentalEffort_T2_MOSKitt4ME" variable does not meet this requirement. For this reason, we performed non-parametric tests; specifically, Wilcoxon signed-rank tests. Below, we present the normality tests that we performed in our study (Section C.4.1); then, we present the results of the Wilcoxon signed-rank tests (Section C.4.2).

C.4.1 Normality Tests

In order to test the normality of our data, we applied Shapiro-Wilk tests. The results of these tests are shown in Table C.2. As the table shows (in the right-most column), the only variable whose significance is less than 0.05 is "MentalEffort_T2_MOSKitt4ME"; therefore, all of the other seven variables follow a normal distribution and "MentalEffort_T2_MOSKitt4ME" can be considered as non-normal.

This conclusion is also reflected in the eight Q-Q plots that are printed by the *IBM SPSS Statistics 2.0*. In the first seven plots (each of which corresponds to one of the first seven variables), most data points are close to the line y = x, which represents the expected normal distribution. Figure C.6 shows as an example the Q-Q plot of the first variable (i.e., "Time_T1_None").

Tests of Normality

	Kolmogorov-Smirnov[a]			Shapiro-Wilk		
	Statistic	df	Sig.	Statistic	df	Sig.
Time_T1_None	,170	8	,200*	,915	8	,390
Time_T1_MOSKitt4ME	,281	8	,063	,884	8	,207
Time_T2_None	,237	8	,200*	,857	8	,112
Time_T2_MOSKitt4ME	,202	8	,200*	,945	8	,661
MentalEffort_T1_None	,220	8	,200*	,917	8	,408
MentalEffort_T1_MOSKitt4ME	,249	8	,155	,875	8	,168
MentalEffort_T2_None	,204	8	,200*	,829	8	,058
MentalEffort_T2_MOSKitt4ME	,391	8	,001	,641	8	,000

TABLE C.2: Results of the tests of normality

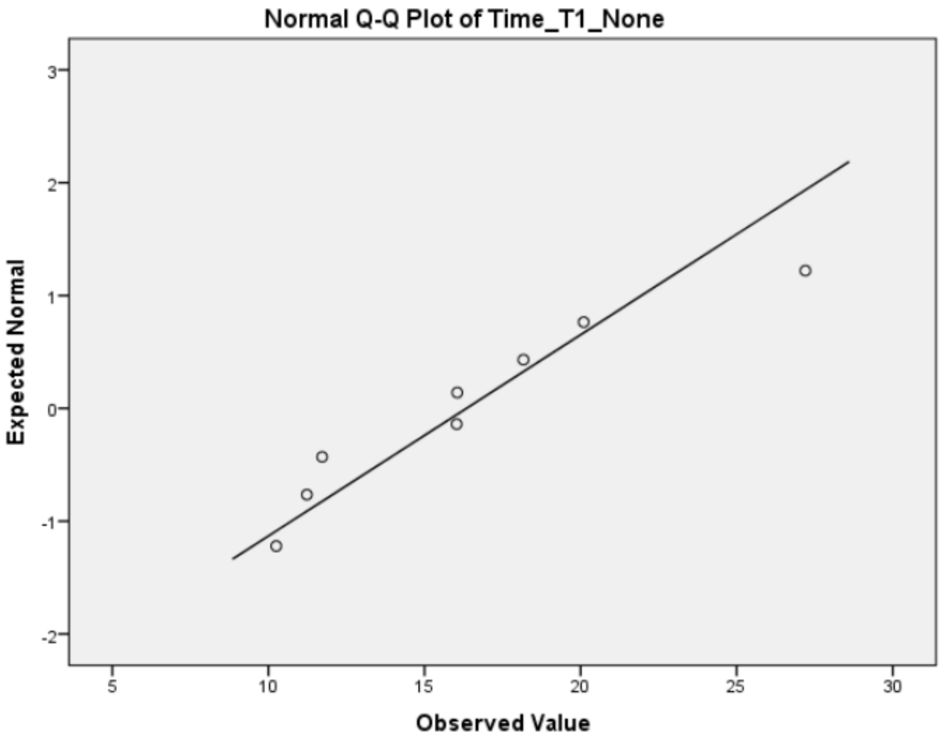
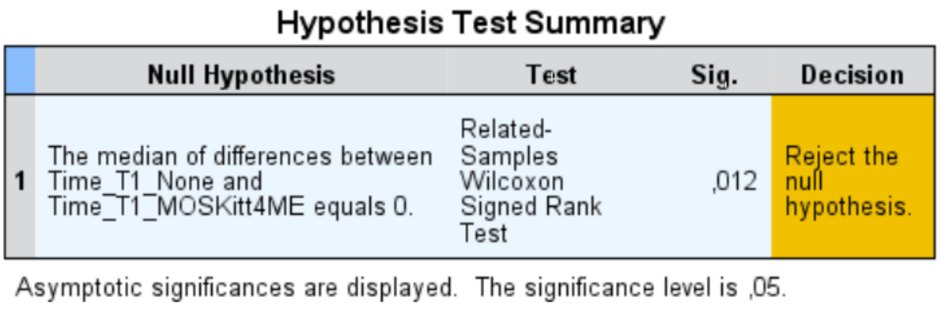


FIGURE C.6: An example of Q-Q plot


Hypothesis Test Summary

	Null Hypothesis	Test	Sig.	Decision
1	The median of differences between Time_T1_None and Time_T1_MOSKitt4ME equals 0.	Related-Samples Wilcoxon Signed Rank Test	,012	Reject the null hypothesis.

Asymptotic significances are displayed. The significance level is ,05.

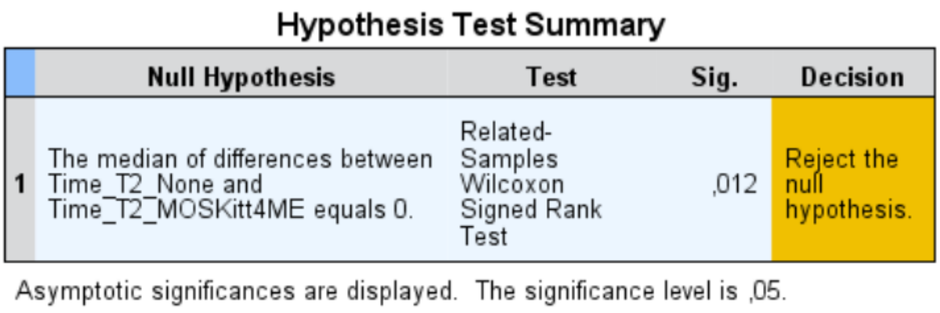
FIGURE C.7: Results of the first Wilcoxon test

C.4.2 Non-parametric Tests

Since one variable (MentalEffort_T2_MOSKitt4ME) does not follow a normal distribution, we performed non-parametric tests; specifically, four Wilcoxon signed-rank tests. Each of these tests compares two variables of the study. The two variables that are involved in a single test measure the same data (i.e., either task completion time or mental effort), relate to the same task (i.e., either method design/implementation or method execution), but they differ on the applied treatment. Thus, the Wilcoxon tests aim to evaluate the significance of the differences between the results obtained in the two treatments of the study.

The first Wilcoxon test evaluates the following two variables: "Time_T1_None" and "Time_T1_MOSKitt4ME". For this test, the summary of results that is provided by the *IBM SPSS Statistics 2.0* is graphically depicted in Figure C.7. As the figure shows, the null hypothesis can be rejected since $p < \alpha$ (where $p = 0.012$ and $\alpha = 0.05$). Thus, the difference in task completion time (between Treatment 1 and Treatment 2) can be considered significant for the task of method design/implementation.

The second Wilcoxon test evaluates the following variables: "Time_T2_None" and "Time_T2_MOSKitt4ME". The summary of results of this test is graphically depicted in Figure C.8. As the figure shows, the null hypothesis can be rejected since the condition $p < \alpha$ (where $p = 0.012$ and $\alpha = 0.05$) is fulfilled. Thus, the difference in task completion time (between Treatment 1 and Treatment 2) can be considered significant for the task of method execution.

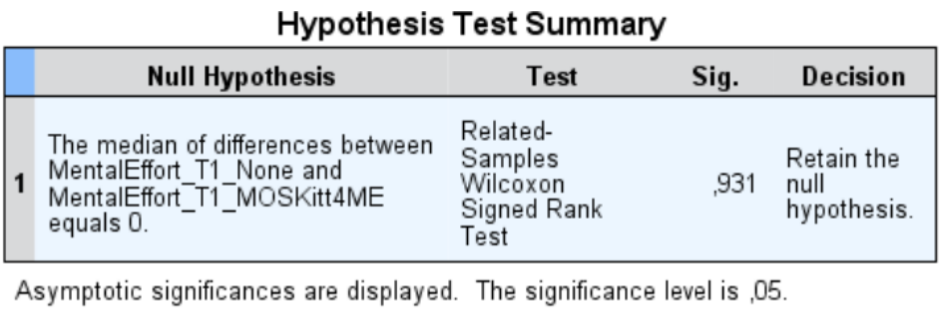


Hypothesis Test Summary

	Null Hypothesis	Test	Sig.	Decision
1	The median of differences between Time_T2_None and Time_T2_MOSKitt4ME equals 0.	Related-Samples Wilcoxon Signed Rank Test	,012	Reject the null hypothesis.

Asymptotic significances are displayed. The significance level is ,05.

FIGURE C.8: Results of the second Wilcoxon test



Hypothesis Test Summary

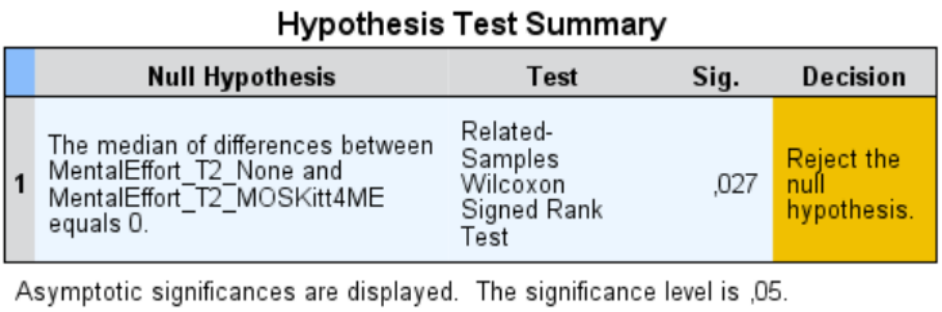
	Null Hypothesis	Test	Sig.	Decision
1	The median of differences between MentalEffort_T1_None and MentalEffort_T1_MOSKitt4ME equals 0.	Related-Samples Wilcoxon Signed Rank Test	,931	Retain the null hypothesis.

Asymptotic significances are displayed. The significance level is ,05.

FIGURE C.9: Results of the third Wilcoxon test

The third Wilcoxon test evaluates the "MentalEffort_T1_None" and the "MentalEffort_T1_MOSKitt4ME" variables. The results are depicted in Figure C.9. In this case, the null hypothesis cannot be rejected because $p > \alpha$ (where $p = 0.931$ and $\alpha = 0.05$). Thus, the difference in mental effort (between Treatment 1 and Treatment 2) cannot be considered statistically significant for the task of method design/implementation.

Finally, the fourth Wilcoxon test evaluates the "MentalEffort_T2_None" and the "MentalEffort_T2_MOSKitt4ME" variables. The summary of results of this test is depicted in Figure C.10. As the figure shows, the null hypothesis can be rejected since the condition $p < \alpha$ (where $p = 0.027$ and $\alpha = 0.05$) is fulfilled. Thus, the difference in mental effort (between Treatment 1 and Treatment 2) can be considered significant for the task of method execution.



Hypothesis Test Summary

	Null Hypothesis	Test	Sig.	Decision
1	The median of differences between MentalEffort_T2_None and MentalEffort_T2_MOSKitt4ME equals 0.	Related-Samples Wilcoxon Signed Rank Test	,027	Reject the null hypothesis.

Asymptotic significances are displayed. The significance level is ,05.

FIGURE C.10: Results of the fourth Wilcoxon test

C.5 Coding Scheme

This section contains the coding scheme that was developed during the phase of data analysis. We used this coding scheme to annotate the transcriptions that were obtained from the Think Aloud sessions. The annotation process resulted in a set of Think Aloud protocols that can be found in the MOSKitt4ME website: `http://users.dsic.upv.es/~mcervera/moskitt4me`.

Category 1. Errors

All of the items of this category follow the pattern *code – description*.

- E1 – Selection of incorrect tool for executing a method task.
- E2 – Creation of an incorrect method element or relationship.
- E3 – Specification of an incorrect property value.
- E4 – Eclipse error not related with MOSKitt4ME.
- E5 – Selection of incorrect GUI component.
- E7 – Performance of an action that is either not related to the task at hand or unnecessary for its proper execution.
- E8 – Performance of an action that is the incorrect way to achieve correct (or partially correct) results within a task.
- E9 – Incorrect transition between tasks of the method.

- E10 – Execution of an action that does not correspond to the experiment task at hand.
- E11 – Incorrect association between a conceptual fragment and a technical fragment.
- E12 – Error occurring due to a MOSKitt4ME bug.
- E13 – Omission of the “clean” command when restarting Eclipse.
- E14 – Confusing a plug-in that is already installed with a software dependency.
- E15 – Creation of an incorrect file.
- E16 – Abandoning method execution.

Category 2. Challenges

All of the items of this category follow the pattern *code – description*. Utterances or actions such as doubts (C2), guessing (C5), postponing analysis of action (S6), and trial and error (S1) may point to challenges.

- CH1 – Management of the plug-ins dependencies without appropriate tool support.
- CH2 – Selection of the appropriate tool to carry out a task.
- CH3 – Properly managing the GUI.
- CH5 – Understanding the separation of method content and processes.
- CH6 – Understanding the relationships between conceptual and technical fragments.
- CH7 – Understanding the concepts of method design, method implementation, and method execution.
- CH8 – Understanding the relationship between the SPEM 2.0 model and the BPMN 2.0 model.
- CH9 – Understanding the semantics of the method.
- CH10 – Understanding the semantics of SPEM 2.0.

- CH11 – Understanding the notion of technical fragment.
- CH12 – Understanding the general problem to solve (in a task of the experiment) and how to solve it.
- CH13 – Fixing problems with the help of the Eclipse error messages.

Category 3. Tasks

All of the items of this category follow the pattern *code – description.*

- T1 – Glossary of terms definition.
- T2 – Business logic design.
- T3 – Database model specification.
- T4 – Database scripts generation.
- T5 – Database scripts revision.

Category 4. Expert Knowledge

All of the items of this category follow the pattern *code – description – example.*

- EK1 – The subject has previous knowledge related to the task – "I am familiar with this tool so I don't need to check ...".
- EK2 – The subject gains knowledge during the task execution – "Now I know that this cannot be done".
- EK3 – The task requires knowledge to be properly carried out – "To resolve this problem I should be familiar with ...".

Category 5. Strategies

All of the items of this category follow the pattern *code – description – example.*

- S1 – Trial and error – "I will try this to see if it works".
- S2 – Revising previous work – "I'd like to check first if I did this correctly".

- S3 – Proposing a solution – “The way to solve the problem is . . . ”.
- S4 – Justifying a proposed solution – “This is the way to go because . . . ”.
- S5 – Retracting a previous solution – “This approach is wrong, what if I ..”.
- S6 – Postponing analysis of action – “I will have to work that out later”.

Category 6. Actions

All of the items of this category follow the pattern *code – description.*

- A1 – Opening a tool.
- A2 – Closing a tool.
- A3 – Clearing the Error Log.
- A4 – Copying a folder.
- A5 – Creating a project.
- A6 – Setting a file name.
- A7 – Selecting a root element.
- A8 – Adding new elements to a model.
- A9 – Editing model elements.
- A10 – Consulting method.
- A11 – Consulting task description.
- A12 – Consulting slides or user manual.
- A13 – Opening a file.
- A14 – Closing a file.
- A15 – Deleting a file.
- A16 – Opening a view.
- A17 – Selecting input model.
- A18 – Selecting destination folder.
- A19 – Opening a perspective.

- A20 – Connecting to a repository.
- A21 – The subject performs a physical action that is not related with the task.
- A22 – Deleting elements from a model.
- A23 – Creating a file.
- A24 – Checking the Error Log.
- A25 – Undoing an action.
- A26 – Applying a filter.
- A27 – Creating a new folder.
- A28 – Changing display mode.
- A29 – Selecting a tool.
- A30 – Launching CASE generation.

Category 7. Comments

All of the items of this category follow the pattern *code – description – example.*

- C1 – Comments about the subject's actions – "I am going to create a new project".
- C2 – Doubts – "I don't know if I opened the tool correctly".
- C3 – Comments showing resolution – "Ok, definitely this is the place where …".
- C4 – Getting a result – "I can see that the list of dependencies is …".
- C5 – Guessing – "I think it must be finished now".
- C6 – Getting information from documentation – "The next task to execute is …".
- C7 – Reading – "Designer … Analyst …".
- C8 – Comments that are not related with the task at hand – "Oh, I must not forget to call my friend".

- C9 – Evaluation of the task or task-situation at a meta-level (i.e., complaints, opinions, etc.) – “It is tiring to talk so much”.
- C10 – Comments related with problems or errors – “The editor does not work”.
- C11 – Comments showing reasoning – “In order to resolve this, I should first ...”.
- C12 – Unintelligible comments.
- C13 – Suggestions – “This component should allow me to ...”.
- C14 – Comments related to the current situation – “I am in the work products tab”.
- C15 – Comments related with tool capabilities – “This editor allows me to ...”.
- C16 – Comments related with tool limitations – “This editor does not allow me to ...”.
- C17 – Comments expressing emotions – “Great!”.
- C18 – Comments expressing regret – “I should have done this before”.
- C19 – Giving an example – “For instance, I could ...”.

Bibliography

[1] Alistair Cockburn. Selecting a project's methodology. *IEEE software*, 17 (4):64–71, 2000.

[2] Robert L Glass. Matching methodology to problem domain. *Communications of the ACM*, 47(5):19–21, 2004.

[3] Brian Henderson-Sellers and Jolita Ralyté. Situational method engineering: State-of-the-art review. *Journal of Universal Computer Science*, 16(3):424–478, 2010.

[4] Fredrik Karlsson and Par J Ågerfalk. Method configuration: adapting to situational characteristics while creating reusable assets. *Information and Software Technology*, 46(9):619–633, 2004.

[5] Sjaak Brinkkemper. Method engineering: engineering of information systems development methods and tools. *Information and software technology*, 38(4):275–280, 1996.

[6] Marko Bajec, Damjan Vavpotič, and Marjan Krisper. Practice-driven approach for creating project-specific software development methods. *Information and Software Technology*, 49(4):345–365, 2007.

[7] Naveen Prakash. Towards a formal definition of methods. *Requirements Engineering*, 2(1):23–50, 1997.

[8] Sjaak Brinkkemper, Motoshi Saeki, and Frank Harmsen. Meta-modelling based assembly techniques for situational method engineering. *Information Systems*, 24(3):209–228, 1999.

[9] Jolita Ralyté, Rébecca Deneckère, and Colette Rolland. Towards a generic model for situational method engineering. In *Advanced Information Systems Engineering*, pages 95–110. Springer, 2003.

[10] Colette Rolland. Method engineering: towards methods as services. *Software Process: Improvement and Practice*, 14(3):143–164, 2009.

[11] Marco Kuhrmann, Daniel Méndez Fernández, and Michaela Tiessler. A mapping study on the feasibility of method engineering. *Journal of Software: Evolution and Process*, pages 1–22, 2014.

[12] Kent Beck. *Extreme programming explained: embrace change.* Addison-Wesley Professional, 2000.

[13] Ken Schwaber and Mike Beedle. *Agile Software Development with Scrum.* Prentice Hall, 2001.

[14] Alistair Cockburn. *Crystal clear: a human-powered methodology for small teams.* Pearson Education, 2004.

[15] Philippe Kruchten. *The rational unified process: an introduction.* Addison-Wesley Professional, 2004.

[16] Barry W. Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61–72, 1988.

[17] Michael Turner. *Microsoft solutions framework essentials: building successful technology solutions.* Microsoft Press, 2006.

[18] OMG. Software & Systems Process Engineering Metamodel (v2.0), 2007.

[19] Alfonso Fuggetta. A classification of CASE technology. *Computer*, 26(12): 25–38, 1993.

[20] Anton Frank Harmsen. *Situational method engineering.* PhD thesis, University of Twente, 1997.

[21] Ali Niknafs and Mohsen Asadi. Towards a process modeling language for method engineering support. In *Computer Science and Information Engineering, 2009 WRI World Congress on*, volume 7, pages 674–681. IEEE, 2009.

[22] Cesar Gonzalez-Perez and Brian Henderson-Sellers. A work product pool approach to methodology specification and enactment. *Journal of Systems and Software*, 81(8):1288–1305, 2008.

[23] Jolita Ralyté and Colette Rolland. An assembly process model for method engineering. In *Advanced information systems engineering*, pages 267–283. Springer, 2001.

[24] PS Seligmann, GM Wijers, and HG Sol. Analyzing the structure of is methodologies, an alternative approach. In *Proceedings of the First Dutch Conference on Information Systems, Amersfoort, The Netherlands, EU*, 1989.

[25] Adrian Iacovelli, Carine Souveyet, and Colette Rolland. Method as a service (MaaS). In *Second International Conference on Research Challenges in Information Science (RCIS)*, pages 371–380. IEEE, 2008.

[26] Brian Henderson-Sellers, Jolita Ralyté, Par Ågerfalk, and Matti Rossi. *Situational Method Engineering*. Springer, 2013.

[27] Marko Bajec. Application of method engineering principles in practice: Lessons learned and prospects for the future. In *Engineering Methods in the Service-Oriented Context*, pages 2–3. Springer, 2011.

[28] Arthur HM Ter Hofstede and TF Verhoef. On the feasibility of situational method engineering. *Information Systems*, 22(6):401–422, 1997.

[29] Juha-Pekka Tolvanen. *Incremental Method Engineering with Modeling Tools: Theoretical Principles and Empirical Evidence*. PhD thesis, University of Jyvaskyla, 1998.

[30] Par J Ågerfalk, Sjaak Brinkkemper, Cesar Gonzalez-Perez, Brian Henderson-Sellers, Fredrik Karlsson, Steven Kelly, and Jolita Ralyté. Modularization constructs in method engineering: towards common ground? In *Situational method engineering: fundamentals and experiences*, pages 359–368. Springer, 2007.

[31] Brian Henderson-Sellers. Method engineering for OO systems development. *Communications of the ACM*, 46(10):73–78, 2003.

[32] Brian Fitzgerald, Nancy L Russo, and Tom O'Kane. Software development method tailoring at motorola. *Communications of the ACM*, 46(4):64–70, 2003.

[33] Kai Wistrand and Fredrik Karlsson. Method components–rationale revealed. In *Advanced Information Systems Engineering*, pages 189–201. Springer, 2004.

[34] Brian Fitzgerald. The use of systems development methodologies in practice: a field study. *Information Systems Journal*, 7(3):201–212, 1997.

[35] Brian Lings and Bjorn Lundell. Method in action and method in tool: some implications for CASE. In *ICEIS 2004*, pages 623–628, 2004.

[36] Brian Henderson-Sellers and MK Serour. Creating a dual-agility method: The value of method engineering. *Journal of Database Management*, 16(4): 1–24, 2005.

[37] Marko Bajec and Damjan Vavpotič. A framework and tool-support for reengineering software development methods. *Informatica*, 19(3):321–344, 2008.

[38] Daya Gupta and Naveen Prakash. Engineering methods from method requirements specifications. *Requirements Engineering*, 6(3):135–160, 2001.

[39] Fredrik Karlsson and Par J Ågerfalk. MC Sandbox: Devising a tool for method-user-centered method configuration. *Information and Software Technology*, 54(5):501–516, 2012.

[40] 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*, pages 1–21. Springer, 1996.

[41] Ali Niknafs and Raman Ramsin. Computer-aided method engineering: an analysis of existing environments. In *Advanced Information Systems Engineering*, pages 525–540. Springer, 2008.

[42] Juha-Pekka Tolvanen and Matti Rossi. Metaedit+: defining and using domain-specific modeling languages and code generators. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 92–93. ACM, 2003.

[43] Parastoo Mohagheghi, Wasif Gilani, Alin Stefanescu, and Miguel A Fernandez. An empirical study of the state of the practice and acceptance of model-driven engineering in four industrial cases. *Empirical Software Engineering*, 18(1):89–116, 2013.

[44] Jeff Kramer. Is abstraction the key to computing? *Communications of the ACM*, 50(4):36–42, 2007.

[45] Fred D Davis. Perceived usefulness, perceived ease of use, and user acceptance of information technology. *MIS quarterly*, pages 319–340, 1989.

[46] Cynthia K. Riemenschneider, Bill C. Hardgrave, and Fred D. Davis. Explaining software developer acceptance of methodologies: a comparison of five theoretical models. *IEEE Transactions on Software Engineering*, 28 (12):1135–1145, 2002.

[47] Viswanath Venkatesh and Fred D Davis. A model of the antecedents of perceived ease of use: Development and test. *Decision sciences*, 27(3):451–481, 1996.

[48] Maarten W Van Someren, Yvonne F Barnard, Jacobijn AC Sandberg, et al. *The think aloud method: A practical guide to modelling cognitive processes*, volume 2. Academic Press London, 1994.

[49] OMG. Business Process Model and Notation (v2.0), 2011.

[50] Alan R Hevner, Salvatore T March, Jinsoo Park, and Sudha Ram. Design science in information systems research. *MIS quarterly*, 28(1):75–105, 2004.

[51] Salvatore T March and Gerald F Smith. Design and natural science research on information technology. *Decision support systems*, 15(4):251–266, 1995.

[52] Vijay Vaishnavi and William Kuechler. Design science research in information systems. `http://desrist.org/design-research-in-information-systems/`, 2004.

[53] J. Bergstra, H. Jonkers, and J. Obbink. A software development model for method engineering. In *Esprit 1984: Status Report of Ongoing Work*, 1985.

[54] Kuldeep Kumar and Richard J Welke. Methodology engineering r: a proposal for situation-specific methodology construction. In *Challenges and*

strategies for research in systems development, pages 257–269. John Wiley & Sons, Inc., 1992.

[55] Kees van Slooten and Sjaak Brinkkemper. A method engineering approach to information systems development. In *Proceedings of the IFIP WG8. 1 Working Conference on Information System Development Process*, pages 167–186. North-Holland Publishing Co., 1993.

[56] Gregor Engels and Stefan Sauer. A meta-method for defining software engineering methods. In Gregor Engels, Claus Lewerentz, Wilhelm Schafer, Andy Schurr, and Bernhard Westfechtel, editors, *Graph transformations and model-driven engineering*, volume 5765 of *Lecture Notes in Computer Science*, pages 411–440. Springer Berlin Heidelberg, 2010.

[57] Philip A Laplante. *What every engineer should know about software engineering*. CRC Press, 2007.

[58] Ivar Jacobson, Grady Booch, James Rumbaugh, James Rumbaugh, and Grady Booch. *The unified software development process*, volume 1. Addison-Wesley Reading, 1999.

[59] Naveen Prakash and SB Goyal. Towards a life cycle for method engineering. In *Proceedings Eleventh International Workshop on Exploring Modeling Methods in Systems Analysis and Design (EMMSAD'07)*, pages 27–36, 2007.

[60] M Leppanen. Conceptual analysis of current me artifacts in terms of coverage: a contextual approach. In *1st International Workshop on Situational Engineering Processes Methods, Techniques and Tools to Support Situation-Specific Requirements Engineering Processes (SREP)*, pages 75–90, 2005.

[61] Frank Harmsen and Sjaak Brinkkemper. Description and manipulation of method fragments for the assembly of situational methods. *Memoranda Informatica*, pages 94–52, 1994.

[62] Chittoor V Ramamoorthy, Atul Prakash, Wei-Tek Tsai, and Yutaka Usuda. Software engineering: problems and perspectives. *Computer*, 17(10):191–209, 1984.

[63] Manuel Bollain and Juan Garbajosa. A metamodel for defining development methodologies. In *Software and Data Technologies*, pages 414–425. Springer, 2009.

[64] Manfred A Jeusfeld, Matthias Jarke, and John Mylopoulos. *Metamodeling for method engineering.* the MIT Press, 2009.

[65] Cesar Gonzalez-Perez, Tom Mcbride, and Brian Henderson-Sellers. A metamodel for assessable software development methodologies. *Software Quality Journal*, 13(2):195–214, 2005.

[66] Brian Henderson-Sellers and Cesar Gonzalez-Perez. A comparison of four process metamodels and the creation of a new generic standard. *Information and software technology*, 47(1):49–65, 2005.

[67] ISO/IEC 24744. Software Engineering: Metamodel for Development Methodologies, 2007.

[68] Brian Henderson-Sellers and Cesar Gonzalez-Perez. Standardizing methodology metamodelling and notation: an ISO exemplar. In Roland Kaschek, Christian Kop, Claudia Steinberger, and Günther Fliedl, editors, *Information Systems and e-Business Technologies*, volume 5 of *Lecture Notes in Business Information Processing*, pages 1–12. Springer Berlin Heidelberg, 2008.

[69] Iván Ruiz-Rube, Juan Manuel Dodero, Manuel Palomo-Duarte, Mercedes Ruiz, and David Gawn. Uses and applications of SPEM process models. A systematic mapping study. *Journal of Software Maintenance and Evolution: Research and Practice*, 1(32):999–1025, 2012.

[70] Daniel Teichroew. Problem statement languages in MIS. In *Management-Informationssysteme*, pages 251–282. Springer, 1971.

[71] Daniel Teichroew and Ernest A Hershey III. PSL/PSA: A computer-aided technique for structured documentation and analysis of information processing systems. *Software Engineering, IEEE Transactions on*, (1):41–48, 1977.

[72] Ahmed Abd El-Rahman Mahdy, Mohamed Mohamed Abd El-Salam Ahmed, and Sherif Mohamed Zahran. Flexible CASE tools for requirements engineering: a benchmarking survey. In *International Conference of Informatics and Systems*, number 2, pages 20–26, 2008.

[73] Albert F Case. Computer-aided software engineering (CASE): technology for improving software development productivity. *ACM SIGMIS Database*, 17(1):35–43, 1985.

[74] Elliot J Chikofsky and Burt L Rubenstein. CASE: reliability engineering for information systems. *Software, IEEE*, 5(2):11–16, 1988.

[75] Carma McClure. *CASE is software automation.* Prentice-Hall, Inc., 1988.

[76] Charles F Martin. MetaCASE: dream or reality. In *Electro/94 International. Conference Proceedings. Combined Volumes.*, pages 195–199. IEEE, 1994.

[77] Jeffrey E Kottemann and Benn R Konsynski. Dynamic metasystems for information systems development. In *Proc. of the 5th Intl. Conf. on Information Systems*, pages 187–204, 1984.

[78] Paul G Sorenson, Jean-Paul Tremblay, and Andrew J McAllister. The metaview system for many specification environments. *IEEE software*, 5 (2):30–38, 1988.

[79] Steven Kelly and Kari Smolander. Evolution and issues in metaCASE. *Information and Software Technology*, 38(4):261–266, 1996.

[80] Pentti Marttiin. Towards flexible process support with a CASE shell. In *Advanced Information Systems Engineering*, pages 14–27. Springer, 1994.

[81] Per Bergsten, J Bubenko, Roland Dahl, Mats Gustafsson, and Lars-Ake Johansson. RAMATIC - a CASE shell for implementation of specific CASE tools. *TEMPORA T6*, 1, 1989.

[82] JP Gray, Anna Liu, and Louise Scott. Issues in software engineering tool construction. *Information and software technology*, 42(2):73–77, 2000.

[83] Jurgen Ebert, Roger Suttenbach, and Ingar Uhe. Meta-CASE in Practice: a Case for KOGGE. In *Advanced Information Systems Engineering*, pages 203–216. Springer, 1997.

[84] Nathaniel Palmer. XML Process Definition Language. In *Encyclopedia of Database Systems*, pages 3601–3601. Springer, 2009.

[85] Wil MP Van der Aalst and Arthur HM Ter Hofstede. YAWL: yet another workflow language. *Information systems*, 30(4):245–275, 2005.

[86] Diane Jordan, John Evdemon, Alexandre Alves, Assaf Arkin, Sid Askary, Charlton Barreto, Ben Bloch, Francisco Curbera, Mark Ford, Yaron Goland, et al. Web services business process execution language version 2.0. *OASIS standard*, 11:11, 2007.

[87] Thomas Allweyer. *BPMN 2.0: introduction to the standard for business process modeling.* BoD–Books on Demand, 2010.

[88] Reda Bendraou, Benoit Combemale, Xavier Crégut, and Marie-Pierre Gervais. Definition of an executable SPEM 2.0. In *Software Engineering Conference, 2007. APSEC 2007. 14th Asia-Pacific*, pages 390–397. IEEE, 2007.

[89] Mario Cervera, Manoli Albert, Victoria Torres, and Vicente Pelechano. The MOSKitt4ME approach: providing process support in a method engineering context. In *Conceptual Modeling*, pages 228–241. Springer, 2012.

[90] Jan Recker, Michael Rosemann, Marta Indulska, and Peter Green. Business process modeling - a comparative analysis. *Journal of the Association for Information Systems*, 10(4):1, 2009.

[91] OMG. Model-Driven Architecture, 2001.

[92] Anneke G Kleppe, Jos B Warmer, and Wim Bast. *MDA explained: the model driven architecture: practice and promise.* Addison-Wesley Professional, 2003.

[93] Stuart Kent. Model driven engineering. In *Integrated formal methods*, pages 286–298. Springer, 2002.

[94] Jean Bézivin. In search of a basic principle for model driven engineering. *Novatica Journal, Special Issue*, 5(2):21–24, 2004.

[95] Jean-Marie Favre. Megamodeling and etymology - a story of words: From MED to MDE via MODEL in five milleniums. In *In Dagstuhl Seminar on Transformation Techniques in Software Engineering, number 05161 in DROPS 04101. IFBI.* Citeseer, 2005.

[96] Frédéric Fondement and Raul Silaghi. Defining model driven engineering processes. In *Third International Workshop in Software Model Engineering (WiSME), held at the 7th International Conference on the Unified Modeling Language (UML).* Citeseer, 2004.

[97] Jon Whittle, John Hutchinson, and Mark Rouncefield. The state of practice in model-driven engineering. *Software, IEEE*, 31(3):79–85, 2014.

[98] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. Model-driven software engineering in practice. *Synthesis Lectures on Software Engineering*, 1(1): 1–182, 2012.

[99] Colin Atkinson and Thomas Kuhne. Model-driven development: a metamodeling foundation. *Software, IEEE*, 20(5):36–41, 2003.

[100] Stephen J Mellor, Tony Clark, and Takao Futagami. Model-driven development: guest editors' introduction. *IEEE software*, 20(5):14–18, 2003.

[101] Jesús Sánchez Cuadrado and Jesús García Molina. Building domain-specific languages for model-driven development. *Software, IEEE*, 24(5):48–55, 2007.

[102] Arie Van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *Sigplan Notices*, 35(6):26–36, 2000.

[103] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: A model transformation tool. *Science of computer programming*, 72(1):31–39, 2008.

[104] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006.

[105] Jochen M Kuster, Shane Sendall, and Michael Wahler. Comparing two model transformation approaches. In *Proc. Workshop on OCL and Model Driven Engineering*, 2004.

[106] Anna Gerber, Michael Lawley, Kerry Raymond, Jim Steel, and Andrew Wood. Transformation: The missing link of MDA. In *Graph Transformation*, pages 90–105. Springer, 2002.

[107] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, volume 45, pages 1–17. Citeseer, 2003.

[108] Jon Oldevik, Tor Neple, Roy Grønmo, Jan Aagedal, and Arne-J Berre. Toward standardised model to text transformations. In *Model Driven Architecture–Foundations and Applications*, pages 239–253. Springer, 2005.

[109] Gordon Blair, Nelly Bencomo, and Robert B France. Models @ run.time. *Computer*, 42(10):22–27, 2009.

[110] Carlos Cetina, Pau Giner, Joan Fons, and Vicente Pelechano. Autonomic computing through reuse of variability models at runtime: The case of smart homes. *Computer*, 42(10):37–43, 2009.

[111] Robert France and Bernhard Rumpe. Model-driven development of complex software: A research roadmap. In *2007 Future of Software Engineering*, pages 37–54. IEEE Computer Society, 2007.

[112] Stephen J Mellor, Marc Balcer, and Ivar Foreword By-Jacoboson. *Executable UML: A foundation for model-driven architectures.* Addison-Wesley Longman Publishing Co., Inc., 2002.

[113] Matjaz B Juric, Benny Mathew, and Poornachandra G Sarang. *Business Process Execution Language for Web Services: An Architect and Developer's Guide to Orchestrating Web Services Using BPEL4WS.* Packt Publishing Ltd, 2006.

[114] Germán Harvey Alférez Salinas. *Achieving Autonomic Web Service Compositions with Models at Runtime.* PhD thesis, Universitat Politècnica de València, 2013.

[115] Jim desRivieres and John Wiegand. Eclipse: A platform for integrating development tools. *IBM Systems Journal*, 43(2):371–383, 2004.

[116] Dan Rubel. The heart of eclipse. *Queue*, 4(8):36–44, 2006.

[117] Frédéric Jouault and Ivan Kurtev. Transforming models with ATL. In *Satellite Events at the MoDELS 2005 Conference*, pages 128–138. Springer, 2006.

[118] Jorg Becker, Ralf Knackstedt, Daniel Pfeiffer, and Christian Janiesch. Configurative method engineering - on the applicability of reference modeling mechanisms in method engineering. *AMCIS 2007 Proceedings*, page 56, 2007.

[119] Mohsen Asadi and Raman Ramsin. Method engineering process patterns. In *Proceedings of the 2nd India software engineering conference*, pages 143–144. ACM, 2009.

[120] Colette Rolland. Method engineering: State-of-the-art survey and research proposal. *SoMeT*, 9:3–21, 2009.

[121] Jolita Ralyté, Colette Rolland, and Rébecca Deneckère. Towards a meta-tool for change-centric method engineering: A typology of generic operators. In *Advanced Information Systems Engineering*, pages 202–218. Springer, 2004.

[122] Jane Webster and Richard T Watson. Analyzing the past to prepare for the future: Writing a literature review. *Management Information Systems Quarterly*, 26(2):3, 2002.

[123] Sjaak Brinkkemper, Motoshi Saeki, and Frank Harmsen. Assembly techniques for method engineering. In *Advanced Information Systems Engineering*, pages 381–400. Springer, 1998.

[124] Anton Frank Harmsen, JN Brinkkemper, and JL Han Oei. *Situational method engineering for information system project approaches.* University of Twente, Department of Computer Science, 1994.

[125] Frank Harmsen and Sjaak Brinkkemper. Design and implementation of a method base management system for a situational CASE environment. In *Software Engineering Conference, 1995. Proceedings., 1995 Asia Pacific*, pages 430–438. IEEE, 1995.

[126] Sjaak Brinkkemper, Motoshi Saeki, and Frank Harmsen. A method engineering language for the description of systems development methods. In *Advanced Information Systems Engineering*, pages 473–476. Springer, 2001.

[127] Peter Pin-Shan Chen. The entity-relationship model - toward a unified view of data. *ACM Transactions on Database Systems (TODS)*, 1(1):9–36, 1976.

[128] Naveen Prakash. On method statics and dynamics. *Information Systems*, 24(8):613–637, 1999.

[129] Jolita Ralyté. Reusing scenario based approaches in requirement engineering methods: CREWS method base. In *Database and Expert Systems Applications, 1999. Proceedings. Tenth International Workshop on*, pages 305–309. IEEE, 1999.

[130] Jolita Ralyté and Colette Rolland. An approach for method reengineering. In *Conceptual Modeling*, pages 471–484. Springer, 2001.

[131] Brian Henderson-Sellers, Cesar Gonzalez-Perez, and Jolita Ralyté. Comparison of method chunks and method fragments for situational method engineering. In *Software Engineering, 2008. ASWEC 2008. 19th Australian Conference on*, pages 479–488. IEEE, 2008.

[132] Colette Rolland, Naveen Prakash, and Adolphe Benjamen. A multi-model view of process modelling. *Requirements Engineering*, 4(4):169–187, 1999.

[133] Jolita Ralyté. Requirements definition for the situational method engineering. In *Engineering Information Systems in the Internet Context*, pages 127–152. Springer, 2002.

[134] Isabelle Mirbel and Jolita Ralyté. Situational method engineering: combining assembly-based and roadmap-driven approaches. *Requirements Engineering*, 11(1):58–78, 2006.

[135] Donald G Firesmith and Brian Henderson-Sellers. *The OPEN process framework: An introduction*. Pearson Education, 2002.

[136] Brian Henderson-Sellers. Process metamodelling and process construction: examples using the OPEN Process Framework (OPF). *Annals of Software Engineering*, 14(1-4):341–362, 2002.

[137] Didar Zowghi, DG Firesmith, and B Henderson-Sellers. Using the OPEN process framework to produce a situation-specific requirements engineering method. *Proceedings of SREP*, 5:29–30, 2005.

[138] Mahdi Fahmideh Gholami, Mohsen Sharifi, and Pooyan Jamshidi. Enhancing the OPEN Process Framework with service-oriented method fragments. *Software and Systems Modeling*, pages 1–30, 2014.

[139] Anat Aharoni and Iris Reinhartz-Berger. A domain engineering approach for situational method engineering. In *Conceptual Modeling*, pages 455–468. Springer, 2008.

[140] Motoshi Saeki. CAME: The first step to automated method engineering. In *Workshop on Process Engineering for Object-Oriented and Component-Based Development, Anaheim, CA*, 2003.

[141] Motoshi Saeki. Configuration management in a method engineering context. In *Advanced Information Systems Engineering*, pages 384–398. Springer, 2006.

[142] Colette Rolland, Carine Souveyet, and Mario Moreno. An approach for defining ways-of-working. *Information Systems*, 20(4):337–359, 1995.

[143] Colette Rolland and Veronique Plihon. Using generic method chunks to generate process models fragments. In *Requirements Engineering, 1996., Proceedings of the Second International Conference on*, pages 173–180. IEEE, 1996.

[144] Jean-Roch Schmitt. Product modeling for requirements engineering process modelling. In *Information System Development Process*, pages 231–245, 1993.

[145] Samira Si-Said, Colette Rolland, and Georges Grosz. MENTOR: a computer aided requirements engineering environment. In *Advanced Information Systems Engineering*, pages 22–43. Springer, 1996.

[146] Veronique Plihon. MENTOR: an environment supporting the construction of methods. In *Software Engineering Conference, 1996. Proceedings., 1996 Asia-Pacific*, page 384. IEEE, 1996.

[147] Juha-Pekka Tolvanen. Metaedit+: domain-specific modeling for full code generation demonstrated [gpce]. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 39–40. ACM, 2004.

[148] Juha-Pekka Tolvanen and Steven Kelly. Metaedit+: defining and using integrated domain-specific modeling languages. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 819–820. ACM, 2009.

[149] Frank Harmsen and Motoshi Saeki. Comparison of four method engineering languages. In *Method Engineering*, pages 209–231. Springer, 1996.

[150] Minna Koskinen and Pentti Marttiin. Process support in MetaCASE: implementing the conceptual basis for enactable process models in MetaEdit+. In *Eighth Conference on Software Engineering Environments*, pages 110–122. IEEE, 1997.

[151] Jolita Ralyté, Colette Rolland, and Mohamed Ben Ayed. An approach for evolution-driven method engineering. In *Information Modeling Methods and Methodologies*, pages 80–101. IGI Global, 2005.

[152] Cesar Gonzalez-Perez. Supporting situational method engineering with ISO/IEC 24744 and the work product pool approach. In *Situational Method Engineering: Fundamentals and Experiences*, pages 7–18. Springer, 2007.

[153] Cesar Gonzalez-Perez. Tools for an extended object modelling environment. In *Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems*, pages 20–23, 2005.

[154] Rébecca Deneckère and Carine Souveyet. Patterns for extending an OO model with temporal features. In *OOIS'98*, pages 201–218. Springer, 1998.

[155] Rébecca Deneckère. *Approche d'extension de méthodes fondée sur l'utilisation de composants génériques.* PhD thesis, Université Panthéon-Sorbonne-Paris I, 2001.

[156] Rébecca Deneckère. Using meta-patterns to construct patterns. In *Object-Oriented Information Systems*, pages 124–134. Springer, 2002.

[157] Fredrik Karlsson and Par J Ågerfalk. Towards structured flexibility in information systems development: Devising a method for method configuration. *Journal of Database Management*, 20:51–75, 2009.

[158] Fredrik Karlsson and Kai Wistrand. Combining method engineering with activity theory: theoretical grounding of the method component concept. *European Journal of Information Systems*, 15(1):82–90, 2006.

[159] Perdita Stevens and Rob J Pooley. *Using UML: software engineering with objects and components.* Pearson Education, 2006.

[160] Fredrik Karlsson and Par J Ågerfalk. Method-user-centred method configuration. In *Proceedings of the first international workshop on situational requirements engineering processes: methods, techniques and tools to support situation-specific requirements engineering processes (SREP'05), Paris France*, pages 31–43, 2005.

[161] Julio Ariel Hurtado, María Cecilia Bastarrica, Sergio F Ochoa, and Jocelyn Simmonds. MDE software process lines in small companies. *Journal of Systems and Software*, 86(5):1153–1171, 2013.

[162] Motoshi Saeki. Role of model transformation in method engineering. In *Advanced Information Systems Engineering*, pages 626–642. Springer, 2002.

[163] Fethi Calisir and Ferah Calisir. The relation of interface usability characteristics, perceived usefulness, and perceived ease of use to end-user satisfaction with enterprise resource planning (ERP) systems. *Computers in Human Behavior*, 20(4):505–515, 2004.

[164] Fred D. Davis. User acceptance of information technology: system characteristics, user perceptions and behavioral impacts. *International Journal of Man-Machine Studies*, 38(3):475–487, 1993.

[165] Jen-Her Wu, Shu-Ching Wang, and Li-Min Lin. Mobile computing acceptance factors in the healthcare industry: A structural equation model. *International journal of medical informatics*, 76(1):66–77, 2007.

[166] Barry Oshry. *Seeing systems: Unlocking the mysteries of organizational life.* Berrett-Koehler Publishers, 2007.

[167] Mario Cervera, Manoli Albert, Victoria Torres, and Vicente Pelechano. Model driven method engineering: a case study. Technical report, Centro de Investigación en Métodos de Producción de Software, Universitat Politècnica de València, 2011.

[168] Soichiro Hidaka, Zhenjiang Hu, Kazuhiro Inaba, Hiroyuki Kato, and Keisuke Nakano. GRoundTram: An integrated framework for developing well-behaved bidirectional model transformations. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 480–483. IEEE, 2011.

[169] Christopher M Poskitt, Mike Dodds, Richard F Paige, and Arend Rensink. Towards rigorously faking bidirectional model transformations. In *Proceedings of the Workshop on Analysis of Model Transformations co-located with the 17th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2014), Valencia, Spain*, pages 70–75, 2014.

[170] OMG. Reusable Asset Specification (v2.2), 2005.

[171] Dennis Wagelaar and Ragnhild Van Der Straeten. A comparison of configuration techniques for model transformations. In *Model Driven Architecture–Foundations and Applications*, pages 331–345. Springer, 2006.

[172] Javier Muñoz, Miguel Llacer, and Begoña Bonet. Configuring ATL transformations in MOSKitt. In *Proceedings of the Second International Workshop on Model Transformation with ATL (MtATL)*, 2010.

[173] Eric Freeman, Elisabeth Robson, Bert Bates, and Kathy Sierra. *Head first design patterns.* O'Reilly Media, Inc., 2004.

[174] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software.* Pearson Education, 1994.

[175] Eclipse Process Framework (EPF) Composer . Installation, Introduction, Tutorial and Manual . `http://www.eclipse.org/epf/general/EPF_Installation_Tutorial_User_Manual.pdf`, 2010.

[176] Juha-Pekka Tolvanen, Matti Rossi, and Hui Liu. Method engineering: current research directions and implications for future research. In *Method Engineering*, pages 296–317. Springer, 1996.

[177] Kenia Sousa, Jean Vanderdonckt, Brian Henderson-Sellers, and Cesar Gonzalez-Perez. Evaluating a graphical notation for modelling software development methodologies. *Journal of Visual Languages & Computing*, 23 (4):195–212, 2012.

[178] Steven Kelly and Matti Rossi. Evaluating method engineer performance: an error classification and preliminary empirical study. *Australasian Journal of Information Systems*, 6(1), 1998.

[179] Noureddine Kerzazi and Mathieu Lavallee. Inquiry on usability of two software process modeling systems using iso/iec 9241. In *Electrical and Computer Engineering (CCECE), 2011 24th Canadian Conference on*, pages 000773–000776. IEEE, 2011.

[180] Asif Qumer and Brian Henderson-Sellers. An evaluation of the degree of agility in six agile methods and its applicability for method engineering. *Information and Software Technology*, 50(4):280–295, 2008.

[181] Fredrik Karlsson. A wiki-based approach to method tailoring. In *Proceedings of the 3rd International Conference on the Pragmatic Web: Innovating the Interactive Society*, pages 13–22. ACM, 2008.

[182] Valeria Seidita, Massimo Cossentino, and Salvatore Gaglio. Adapting PASSI to support a goal oriented approach: a situational method engineering experiment. In *Proc. of the Fifth European workshop on Multi-Agent Systems (EUMAS'07)*, 2007.

[183] Kasper Hornbæk. Current practice in measuring usability: Challenges to usability studies and research. *International journal of human-computer studies*, 64(2):79–102, 2006.

[184] Younghwa Lee, Kenneth A Kozar, and Kai RT Larsen. The technology acceptance model: past, present, and future. *Communications of the Association for Information Systems*, 12(1):50, 2003.

[185] Raquel Benbunan-Fich. Using protocol analysis to evaluate the usability of a commercial web site. *Information & Management*, 39(2):151–163, 2001.

[186] Per Runeson and Martin Host. Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering*, 14 (2):131–164, 2009.

[187] Ron D Henderson, Mike C Smith, John Podd, and Hugo Varela-Alvarez. A comparison of the four prominent user-based methods for evaluating the usability of computer software. *Ergonomics*, 38(10):2030–2044, 1995.

[188] Timothy C Lethbridge, Susan Elliott Sim, and Janice Singer. Studying software engineers: Data collection techniques for software field studies. *Empirical software engineering*, 10(3):311–341, 2005.

[189] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Bjöorn Regnell, and Anders Wesslén. *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers, Norwell, MA, USA, 2000. ISBN 0-7923-8682-5.

[190] Stephen Owen, Pearl Brereton, and David Budgen. Protocol analysis: a neglected practice. *Communications of the ACM*, 49(2):117–122, 2006.

[191] John S. Gero and Thomas Mc Neill. An approach to the analysis of design protocols. *Design Studies*, 19(1):21 – 61, 1998.

[192] Barney G. Glaser and Anselm L. Strauss. *The Discovery of Grounded Theory: Strategies for Qualitative Research.* Aldine de Gruyter, New York, NY, 1967.

[193] Klaus Pohl, Günter Böckle, and Frank van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer-Verlag New York, Inc., 2005.

[194] Elena Kornyshova, Rébecca Deneckère, and Colette Rolland. Method families concept: application to decision-making methods. In *Enterprise, Business-Process and Information Systems Modeling*, pages 413–427. Springer, 2011.

[195] Mohsen Asadi, Bardia Mohabbati, Dragan Gaševic, Ebrahim Bagheri, and Marek Hatala. Developing semantically-enabled families of method-oriented architectures. *International Journal of Information System Modeling and Design (IJISMD)*, 3(4):1–26, 2012.

[196] Gwladys Guzelian and Corine Cauvet. SO2M: Towards a service-oriented approach for method engineering. In *Proceedings of the 2007 World Congress in Computer Science, Computer Engineering and Applied Computing*, volume 7, 2007.

[197] Corine Cauvet. Method engineering: a service-oriented approach. In *Intentional Perspectives on Information Systems Engineering*, pages 335–354. Springer, 2010.

[198] Mark Turner, David Budgen, and Pearl Brereton. Turning software into a service. *Computer.*, 36(10):38–44, 2003.

[199] Frédéric Jouault, Bert Vanhooff, Hugo Bruneliere, Guillaume Doux, Yolande Berbers, and Jean Bézivin. Inter-DSL coordination support by combining megamodeling and model weaving. In *Proceedings of the ACM Symposium on Applied Computing*, pages 2011–2018. ACM, 2010.

[200] Jean Bézivin, Frédéric Jouault, and Patrick Valduriez. On the need for megamodels. In *Proceedings of the OOPSLA/GPCE: Best Practices for Model-Driven Software Development workshop, 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2004.

[201] Jean Bézivin, Frédéric Jouault, Peter Rosenthal, and Patrick Valduriez. Modeling in the large and modeling in the small. In *Model Driven Architecture*, pages 33–46. Springer, 2005.

[202] Freddy Allilaire, Jean Bézivin, Hugo Brunelière, and Frédéric Jouault. Global model management in Eclipse GMT/AM3. In *Proceedings of the Eclipse Technology eXchange (eTX) workshop at ECOOP*, 2006.

[203] Ian Sommerville and Gerald Kotonya. *Requirements engineering: processes and techniques*. John Wiley & Sons, Inc., 1998.

[204] CMMI Institute. Capability Maturity Model Integration (v1.3), 2010.

[205] Kamal Zuhairi Zamli and Nor Ashidi Mat Isa. A survey and analysis of process modeling languages. *Malaysian Journal of Computer Science*, 17 (2), 2004.