Automating Object-Oriented Software Development Methods

Bedir Tekinerdogan[1], Motoshi Saeki[2], Gerson Sunye[3],
Pim van den Broek[1], Pavel Hruby[4]

[1]University of Twente, Department of Computer Science, TRESE group,
P.O. Box 217, 7500 AE Enschede, The Netherlands.
{bedir, pimvdb}@cs.utwente.nl, http://wwwtrese.cs.utwente.nl
[2]Tokyo Institute of Technology, Department of Computer Science, Tokyo 152-8552, Japan,
saeki@se.cs.titech.ac.jp, http://www.se.cs.titech.ac.jp
[3] IRISA, Triskell Research Group, Campus de Beaulieu, 35 042 Rennes,
http://gerson.sunye.free.fr
[4] Navision Software a/s, Frydenlunds Allé 6, 2950 Vedbaek, Denmark
ph@navision.com, http://www.navision.com

Abstract. Current software projects have generally to deal with producing and managing large and complex software products. It is generally believed that applying software development methods are useful in coping with this complexity and for supporting quality. As such numerous object-oriented software development methods have been defined. Nevertheless, methods often provide a complexity by their own due to their large number of artifacts, method rules and their complicated processes. We think that automation of software development methods is a valuable support for the software engineer in coping with this complexity and for improving quality. This paper presents a summary and a discussion of the ideas that were raised during the workshop on automating object-oriented software development methods.

1. Introduction

Numerous object-oriented software development methods exist in the literature. Most popular methods have a general character, but some methods, like real-time system design, are targeted at specific application domains. Some methods are specifically defined for a given phase in the life cycle of software development, such as requirement analysis or domain analysis. It is generally accepted that these methods are useful for developing high-quality software.

Most methods include a number of heuristic rules, which are needed to produce or refine different artifacts. Moreover, the rules are structured in different ways, leading to different software development processes. Although useful, applying methods is a complex issue, and does not necessarily lead to effective and efficient software development. Automated support for object-oriented methods will decrease this complexity, increase reusability, and provide better support for adaptability, customizability and continuous improvement. Unfortunately, apart from the many environments with diagram editors and visualization tools, existing object-oriented methods are basically described in separate handbooks and manuals. Complete and integrated tools, which support the entire life cycle, are not yet present in practice.

This workshop aimed to identify the fundamental problems of automating methods and to explore the mechanisms for constructing case tools that provide full support for methods. The initial topics of interest were the following:

- ***Meta-models for software development methods***
  - How to model software and management artifacts?
  - Which meta-models are needed?
  - Development process patterns.
- ***Active rule/process support for methods***
  - How to formalize heuristic rules of methods?
  - How to integrate rules in case tools.
  - How to formalize process of methods.
- ***Method engineering***
  - Tailoring and composing methods.
  - Refinement of methods to projects.
  - Inconsistencies in method integration.
- ***Case tools for method generation***
  - Experiences with meta-case tools.
  - Design of meta-case tools.
- ***Automated support for quality reasoning***
  - Tools for quality management
  - Automated support for alternatives selection.
- ***Existing case tools***
  - Overview/comparison of existing tools with respect to method support
  - Extensions to existing case tools

In the following sections we will report on the ideas that were developed at this workshop. To understand the context, we will first explain the basic elements of a method in section 2 followed by the rationale for applying a method in section 3. Section 4 will present the rationale for automating methods. In section 5 we will provide the program of the workshop and present the categorization and discussion on the papers. Section 6 presents the discussions and the ideas that were developed during the workshop. We will conclude in section 7.

2. What is a Method?

In order to automate methods we first need to understand the basic elements of methods. Figure 1 represents a methodological framework for software development, which consists of four basic layers. The application layer represents the software product being developed using this methodological framework. The method layer includes process descriptions, notations, rules and hints to build the application with the existing computation models. The computation models represent the basic building blocks of the application and include the object-oriented features like objects, classes, messages and inheritance. The tools layer provides tools to support the horizontal layers, like dedicated compilers and CASE tools.

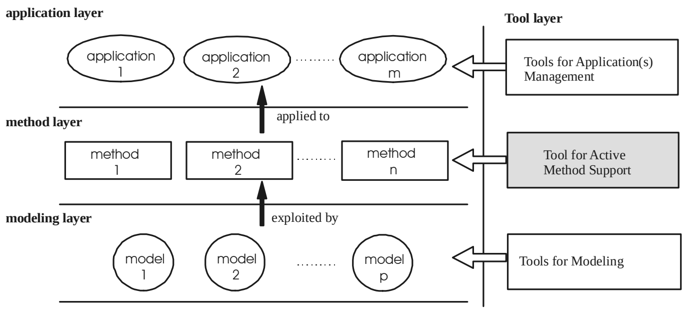

Figure 1. *Methodological framework for software development (adapted from [22])*

Using this methodological framework we can define a software development method in terms of the following aspects:

Artifact types

Artifact types are descriptive forms that the software engineer can utilize for producing artifacts. In this sense, artifact types reflect the properties of the artifacts in the system. For example, the Unified Process [14] provides artifact types for use cases, classes, associations, attributes, inheritance relations and state-charts. Artifact types are represented basically using textual or graphical representations. Artifact types include descriptions of the models in the modeling layer in Figure 2. In addition to these, the method itself may define intermediate or subsidiary artifact types to produce the final software products. An intermediate artifact type in, for example, OMT [19] is the artifact type *Tentative Class*, which describes the entities that are potentially an artifact *Class*, but which may later be eliminated or transformed to the artifact *Attribute*.

Method rules

Method rules aim at identifying, eliminating and verifying the artifacts. Most methods define rules in an informal manner. Nevertheless, method rules can be expressed using conditional statements in the form IF <condition> THEN <consequent> [23]. The consequent part may typically be a selection, elimination or an update action. For example, the Unified Process advises the following (selection) rule to identify classes:

IF an entity in a use case model is relevant
THEN select it as a class

In general, most rules are *heuristic* rules [18]; they support the identification of the solution but there is actually no guarantee that the solution can be found by anybody at anytime by applying the corresponding heuristic rules. The heuristic rules are generally built up over a period of time, as experience is gained in using the method in a wider domain. The application of the heuristic rules depends on the interpretation of the engineer, which may differ because of the different backgrounds and experiences of the engineers. Opposite to heuristic rules are algorithmic rules, which are derived from the concept of algorithm. An algorithm is a unique representation of operations, which will lead to clearly described result. An algorithmic rule is a rule, which can be transformed to an algorithm. Every rule that cannot be transformed to an algorithmic rule is a heuristic rule. Algorithmic rules work best in a predictable and limited environment and where there is full knowledge of all contingencies. Algorithmic rules fail however in unpredictable environments which contain uncertainty, change or competition. In general the gross of the rules in current software development methods are heuristic rules.

Software process

Very often, the term *process* is used to indicate the overall elements that are included in a method, that is, the set of method rules, activities, and practices used to produce and maintain software products. Sometimes the term process is also used as a synonym for the term method. We make an explicit distinction between method and process. In the given methodological framework of Figure 1 a process is part of a method. In this context, we adopt the definition of a process as a (partially) ordered set of actions for achieving a certain goal [10]. The actions of a process are typically the method rules for accessing the artifacts. Process actions can be causally ordered, which represents the time-dependent relations between the various process steps. We adopt the currently accepted term *workflow* to indicate such an ordering [14]. Workflows in software development are, for example, analysis, design, implementation and test. Formerly, this logical ordering of the process actions was also called *phase*. Currently, the term *phase* is more and more used to define time-related aspects such as milestones and iterations [14].

To support the understanding of software processes and improve the quality we may provide different models of processes [1]. Several process models have been proposed, including the traditional waterfall model and the spiral model, which have been often criticized because of the rigid order of the process steps. Recently, more advanced process models such as the Rational Unified Process [16] and the Unified Software Development Process [10] have been proposed.

Software development methods differ in the adopted artifact types, the corresponding method rules and the process that is enforced for applying the method rules. Consequently, automated support for methods can thus basically concern automating artifact management, automating method rules and/or automating the development process.

3. Rationale for Utilizing methods

It is generally believed that the application of methods plays an important role in developing quality software products. The following are the fundamental technical reasons for this.

First, a method provides the designer with a set of guidelines in producing the artifact and its verification against the requirements in the problem statement. This is particularly important for the inexperienced designer who needs assistance to capture the essential aspects of the design. From experimental studies it follows that experienced designers may often follow an opportunistic approach, but that is less effective for inexperienced designers who are not familiar with the problem domain [1][24]. A method directs the designer to produce the right artifact.

Second, since methods formalize certain procedures of design and externalize design thinking, they help to avoid the occurrence of overlooked issues in the design and tend to widen the search for appropriate solutions by encouraging and enabling the designer to think beyond the first solution that comes to mind.

Third, design methods help to provide logical consistency among the different processes and phases in design. This is particularly important for the design of large and complex systems, which is produced by a large team of designers. A design method provides a set of common standards, criteria and goals for the team members.

Fourth, design methods help to reduce possible errors in design and provide heuristic rules for evaluating design decisions.

Finally, mainly from the organizational point of view, a method helps to identify important progress milestones. This information is necessary to control and coordinate the different phases in design.

A method is mainly necessary for structuring the process in producing large scale and complex systems that involve high costs. Motivation for design methods can thus be summarized as directing the designer, widening possible number of design solutions, providing consistency among design processes, reducing errors in design and identifying important milestones.

4. Rationale for Automating Methods

Although methods may include the right process, artifact types and method rules, applying methods may not be trivial at all. Currently, software development is a human-intensive process in which methods are designed and applied by humans with their inherent limitations, who can cope with a limited degree of complexity. Software development is a problem-solving process in which the application of methods is a complex issue. The complexity is firstly caused by the complexity of the problems that need to be solved and secondly by the complexity of the methods themselves. Currently, a valuable and practical method usually includes over dozens of artifact types each corresponding with many method rules that are linked together in a complicated process, which is all together not easy to grasp for the individual mind. In addition, these aspects may also not be explicitly described in the methods and likewise increase complexity. As such, applying the method may be cumbersome, which will directly impact the artifacts that are being produced.

Automating the software development methods can be considered as a viable solution to managing the complexity of the application of methods. Automating the methods will reduce the labor time and eliminate the source of errors in applying the method [7]. In addition, as a matter of fact, many activities in methods do not require specific and/or advanced skills and basically consists of routine work. It may then be worthwhile to automate all the activities so that the software engineer can focus on more conceptual issues. Naturally, there may also be activities that are hard to automate or even impossible for automation, e.g. forming concepts may be one candidate for this.

The software engineering community has an intrinsic tendency towards automating processes and providing tools to cope with the complexity. The so-called Computer Aided Software Engineering (CASE) tools basically aim at automating the various activities in the software development process. Automating methods essentially means that we need to build CASE tools for supporting the application of methods. This is shown in Figure 2 through the gray rectangle in the tool layer.

Automation is inherent to software engineering since it basically automates the solutions for the real world problems. For this purpose, in the beginning of software engineering the major tool was the programming language itself. This was followed by compilers, editors, debuggers, and interpreters. Until the middle of 1980s, tools were developed mainly for the lower level phases of the life cycle. With the exception of general purpose editing facilities, almost no support was provided for the higher level phases. With the advent of interactive graphic tools automated support for graphical design notations appeared on the market in the late 1980s. A collection of related tools is usually called an *environment* [12]. Unfortunately, complete and integrated tools that support the entire life cycle are not yet present in practice. This workshop aimed to identify the problems in these issues and tried to come up with some reusable solutions.

5. Meta-Modeling

Engineers build models to better understand the systems that are being developed [6]. In a similar way, to understand existing models we may provide models of these as well. This activity is called *meta-modeling*. Meta-models are thus abstractions of a set of existing models. They can be used to understand the relationships of the concepts in different modeling languages, for comparing and evaluating different models, for providing interoperability among different tools, or as conceptual schemas for modeling CASE tools and repositories.

To understand software development methods we may thus need to provide models of methods. An example of a model for software development methods is the model in Figure 1. Method-modeling is typically an activity of *method engineering*, which is defined as an engineering discipline for designing, constructing and adapting methods, techniques and tools for the development of information systems [21].

To automate methods both method-engineering and meta-modeling can be applied. CASE tools can be developed for supporting a single method. However, since it is generally difficult to define an ideal method for all application domains and all processes, most CASE environments need to support several methods. To be able to support multiple methods, modern CASE environments basically adopt meta-models of these methods, which can be tailored by method designers. A typical example is the meta-model of the Unified Modeling Language (UML) [6]. The quality of meta-models basically depends on the scope of the models it can describe and its adaptability and extensibility with future requirements. Providing meta-models of existing models is not a trivial task, and method engineering knowledge may provide systematic activities to do this properly.

In the same way that meta-models describe models in a particular language, meta-meta-models express meta-models. To express these ideas the *four-level architecture* [5] has been accepted as an architectural framework for model, meta-models and meta-meta-models. This architecture is shown in Figure 2.

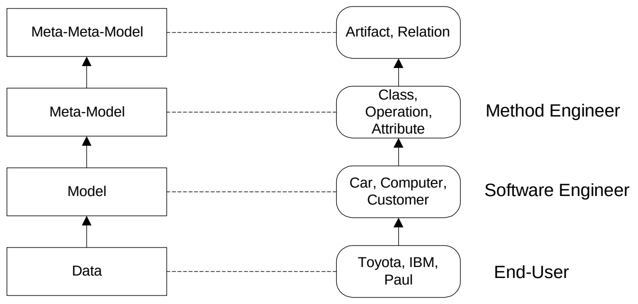

Figure 2. *The four level architecture for modeling*

Hereby the rectangles represent the model layers, whereas the rounded rectangles represent the instantiations of these models.

6. Workshop Program

The workshop topics were related to the background as presented in the previous sections. We have received 14 papers from varying topics, which we have classified into five groups. These papers were shortly presented during the morning. In addition to the presentations, the authors had the opportunity to hang up posters in the room, which were presented off-line during the breaks. In the following we first present the morning program together with a short summary and a discussion of each session. The sessions actually provide a refinement of the framework in Figure 2.

6.1 Refining the four-level architecture

9:00-9:20 Introduction, Bedir Tekinerdogan

This presentation basically discussed the goals of the workshop and the basic elements of methods, the rationale for automating methods and a categorization of the submitted papers.

9:20-10:00 Group 1: ***Meta-Modeling,*** Chair: Motoshi Saeki

- *Medical Reports through Meta-Modeling Techniques: MétaGen in the medical domain*, N. Revault, B. Huet
- *Towards a Tool for Class Diagram Construction and Evolution*, M. Dao, M. Huchard, H. Leblanc, T. Libourel, C. Roume
- *Using UML Profiles: A Case Study*, L. Fuentes, A. Vallecillo
- *Abstraction Levels in Composition*, M. Glandrup

In section 5 we have seen the importance of meta-modeling for designing methods. In this Meta-Modeling session the first two papers concern the application of meta-models while the latter two discuss various aspects of meta-models. Figure 3 summarizes the map of the discussions in the session, and in addition can be considered as a refinement to the four-layer modeling framework in Figure 2.

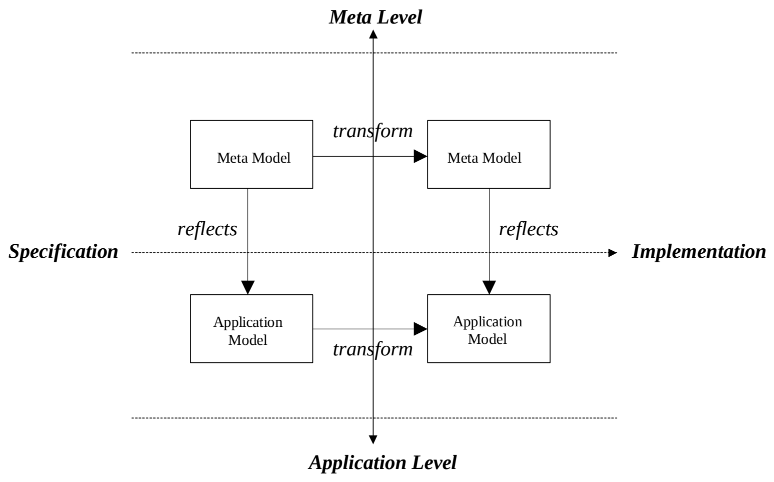

Figure 3. *Aspects of meta-models*

The papers in this session consider two levels of reflection relations: meta-model and application model (meta-model layer and model layer in Figure 2 respectively). The paper of Revault & Huet pointed out that at one level in the meta-modeling architecture different models exist that are transformed to other models at the same level. In addition, they make an explicit distinction between specification and implementation of the models. To develop application programs (semi-) automatically we need to model the transformation of meta-models from specification to implementation. Dao et. al. proposed automated CASE tool generation from a meta-model description so that the construction and the evolution of application models can be supported by manipulating and analyzing the meta-model. The other two papers presented by Glandrup, and Fuentes & Vallecillo, discussed several viewpoints of meta-models themselves. The former captured meta-models from a compositional view and set up five abstraction levels of meta-model composition; behavior, artifact, structure, expression power and expression language. The latter one discussed meta-models from UML profile view and proposed the concepts of basic model and composite one in UML profiles to define meta-models.

10:00-10:30 Group 2: ***Automatic Transformation of Models***, Chair: Gerson Sunyé

- *Automatic Code Generation using an Aspect Oriented Framework,* O. Aldawoud, A. Bader, E. Tzilla
- *Automatic Transformation of Conceptual Models into Design Models,* J. Said, E. Steegmans

This session focuses more on the transformation of models within one level of the meta-modeling architecture of Figure 2.
Hereby, basically two topics were addressed: separation of concerns and traceability. The first paper tries to formally separate the basic algorithm from special purpose concerns such as persistence, synchronization, real-time constraints, etc. This separation allows for the locality of different kinds of functionalities in the programs, making them easier to write, understand, and modify. Hereby, a method for using the separation of concerns at the design level is presented. Their work uses the different UML views to express different concerns. More precisely, they use statecharts to express the concurrency of a system and generate the specialization code for an application framework. Traceability is the degree to which a relationship can be established between two or more models during the development process. Said and Steegmans introduced a Java framework, which helps the development of transformational components, used to translate models from analysis to design. Since this framework can keep a trace of the transformed elements, it keeps traceability dependencies between software development activities.

10:30-11:00 Break
11:00-11:20 Group 3: ***Automatic Support for Patterns***, Chair: Gerson Sunyé

- *Meta-Modeling Design Patterns: Application to pattern detection and code synthesis*, H. Albin-Amiot, Y. Gueheneuc
- Object-Oriented Modeling of Software Patterns and Support Tool, T. Kobayashi

Within one model one may identify patterns of models and patterns of transformations. This session focused on tool support for Design Patterns, which has been recently the subject of several research efforts. The goal of these tools is to help designers in several ways, using different approaches, such as code generation, validation and recognition. Automatic code generation focuses on automatically generating code from design patterns, which likewise releases designers from the implementation burden. Validation ensures that pattern constraints are respected, and since this may be easily overlooked automation may play an important supporting role. Finally, the recognition of pattern instances within source code avoid them to get lost after they have been implemented.

Independently of the approach it supports, a pattern tool must answer to at least two questions: (i) how (and what parts of) the definition of the pattern definition is represented and (ii) how a pattern instance is implemented/recognized.

The tool presented by Albin-Amiot & Yann-Gaël Guéhéneuc uses a Java framework in order to represent the structure of a pattern in terms of Entities (essentially, classes and interfaces) and Elements (associations, methods and fields). In addition, some precise behavior (e.g. delegation) is represented by a Java class. Pattern methods are represented by a declarative description. Once a pattern is precisely represented in the framework, it is used to generate and recognize pattern instances. A similar tool was introduced by Takashi Kobayashi where design patterns are also represented by a Java framework. The representation of a pattern is used by a class-diagram editor, which allows instances of different patterns to be merged.

11:20-11:40 Group 4: **Formal approaches/verification**, Chair: Pim van den Broek

- *Prototype Execution of Independently Constructed Object-Oriented Analysis Model*, T. Aoki, T. Katayama
- *Regulating Software Development Process by Formal Contracts,* C. Pons, G. Baum

Generally, the transformation between the different models is required to be correct. This session focused on automating this verification and validation of the transformation of the models.

In the first paper, the authors propose a formal approach for object-oriented analysis modeling, consisting of formal analysis models, unification of these models, prototype execution of the resulting model, and a prototyping environment. It is shown how the analysis models are formalized, how they are unified into the unified model, and how prototyping execution of the unified model is performed. The purpose of the prototype execution is to ensure the validity of the constructed analysis model. To ensure that the constructed analysis model is correct, it should be verified, which is costly. Therefore the model is validated by prototype execution, and then verified. The prototype execution of the constructed analysis model is done with the functional programming language ML, whose higher order capabilities are useful for modeling application domains.

In the second paper, the authors propose to apply the notion of formal contract to the object-oriented software development process itself. This means that the software development process involves a number of agents (the development team and the software artifacts) carrying out actions with the goal of building a software system that meets the user requirements. Contracts can be used to reason about correctness of the development process and to compare the capabilities of various groupings of agents (coalitions) in order to accomplish a particular contract. The originality of process contracts resides in the fact that software developers are incorporated into the formalism as agents (or coalitions of agents) who make decisions and have responsibilities. Traditional correctness reasoning can be used to show that a coalition of agents achieves a particular goal. Single contracts are analyzed from the point of view of different coalitions with the weakest precondition formalism.

11:40-12:20 group 5: ***Process Support/Modeling***, Chair: Bedir Tekinerdogan

- *Empowering the Interdependence between the Software Architecture and Development Process*, C. Wege
- *Knowledge-Based Techniques to Support Reuse in Vertical Markets*, E. Paesschen
- *HyperCase- Case Tool which Supports the Entire Life Cycle of OODPM*, O. Drori
- *Convergent Architecture Software Development Process*, G. Hillenbrand

This session focused on the concerns in process modeling and process support. In the first paper, Wege observes that the evolution of software artifacts may require the adaptation of the software development process. This may especially the case in the case of software architecture design, which has the largest impact on the overall software development process and which is generally followed by an analysis and design phase. Sometimes, like in Extreme Programming [3], even a constant architecture evolution may be required and it is important to interrelate the changes of the process to software architecture. Wege states that this interdependence between the software architecture and the development process should be made explicit and proposes to provide tool support for this.

Paesschen reflects on transformational and evolutionary dependencies of artifacts in the software development process, such as for example, the dependency between analysis and design. The interesting aspect here is, firstly that artifacts are structurally related, and secondly they may evolve independently. To provide the consistency it is required that the evolution of related artifacts are synchronized. In her paper she specifically focuses on the interdependence between domain models and framework code, and claims that currently the evolution link between the two is implicit but should be captured as knowledge to provide automated support for this. She suggests the development of an expert system that applies this knowledge to provide an explicit coupling between domain models and frameworks.

The last two papers in this session aim to provide tool support for the entire life cycle of the software development process. Drori basically points to the management and control of the various method elements in automating software development methods. He presents a tool called HyperCASE that assumes that the developer already uses a set of tools, and which are structured and managed.

Hillenbrand proposes to apply the so-called convergent architecture software development process that is based on convergent engineering, which aims a convergence between the business domain and the software domain. In the paper the process and the corresponding tool is shortly described.

12:20-12:30 Wrap-up morning session
The morning program ended with a short wrap-up session.

6.2 Preparing Discussions

After the presentations in the morning and the lunch, the program for the afternoon was as follows:

14:00-14:30 Preparing discussions, Motoshi Saeki
In this afternoon session we had planned to identify the important topics that the participants preferred to discuss and that could be considered as a refinement of the ideas that were presented or identified during the morning. Based on the morning presentations and interests of the participants, the following categories were selected as discussions topics:

1. Methodology, which would focus on methods and method engineering techniques
2. Quality, whereby the quality concerns in applying and automating methods were relevant.
3. Meta-Models, which focused on defining meta-models for automating methods.

The basic goal for the discussions was a lively discussion and full information extraction. For this we proposed to utilize so-called index cards in which the following process would be followed: (1) Each member gets 5 index cards (2) On each index card every member writes a question that (s)he thinks is important (3) When everybody has finished writing the questions all the index cards are put on the table (4) Each time randomly an index card is picked up and the question is read by one person (5) The group discusses about the question and categorizes the question. After this, the next person gets the question, reads it and the group categorizes the question, until all index cards have been ordered and categorized. (6) The group tries to find answers for the questions in the different sub-categories, preferably by giving concrete examples.

The subsequent program was as follows:
14:30-15:30 Discussion
15:30-16:00 BREAK
16:00-17:00 Discussion
17:00-17:30 Presentations of the conclusions of the separate groups

6.2 Discussion Results

Methodologies

Automating methods requires a thorough understanding of methods and as such this group focused on the important aspects of software development methods.

The first observation is that different methods may be required for developing different applications and a considerable number of methods have been introduced for various purposes. The problem is that there is actually no universal method for each application and existing methods have been designed for as much as wide range of applications. Nevertheless, they may fail for individual applications. The best possible way is to develop or tailor a dedicated method for each problem domain, that is, engineer methods. This activity of m*ethod engineering* is defined as an engineering discipline for designing, constructing and adapting methods, techniques and tools for the development of information systems [21].

Before we can apply method-engineering techniques and automate methods, it is first required to select the right method or method parts from the extensive set of methods. For this we need to do apply a systematic approach in which we can utilize techniques of domain analysis methods [2]. Domain analysis aims to select and define the domain of focus, and collect the relevant information to provide a domain model. A domain model provides an explicit representation of the common and variant properties of the systems in the domain. Domain analysis applied to software design methods means that we select and define the set of methods that we are interested in, and develop a *method domain model* that includes the commonality and the variabilities of the different methods in the selected domain.

Domain analysis on methods will lead to the observation that some methods are better able to be automated than others. To denote this difference we introduced the quality concept of ***automatability***. We have defined *automatability of methods* as the degree on which methods can be automated. If we consider that every method consists basically of artifact types, method rules and a process as it is explained in section 2, then the first reason for the lack of automatability may be due to the lack of sufficient number of artifact types, method rules and a process. However, this is not the only reason. While some methods are more rigid and seek for high predictability, other methods have by their nature a very flexible and agile process [9]. Flexible methods are less rigid in applying process actions and rely more on intuition of the persons who are involved in the corresponding process actions.

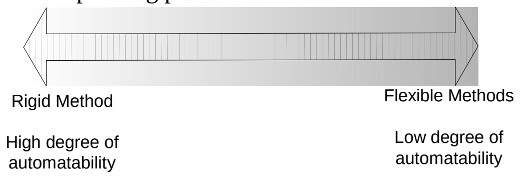

Figure 4. *Automatability of methods with respect to their rigidity*

While flexible methods have a lower automatability degree this does not mean that automation is not possible at all. In this case, the kind of automation will only be different and basically focus on providing supporting tools for the human-centric processes. The bottom line however is that automation is useful for both rigid and flexible methods.

Quality issues in automating methods

Like quality of the artifacts that are produced by software methods we can talk about qualities of methods. The previous section already described a quality factor of *automatability.* In this session, the group has basically focused on the traceability quality factor since this plays an essential role for supporting the automation process and the other quality factors. Traceability requires that the transformational links between the various artifacts must be made explicit and visible to understand their production and to provide automated support for this. The transformation of models exists on various abstraction levels of the four-level architecture in Figure 2. In this session the group focused on transformation of artifacts within one layer, that is, the model layer of Figure 2. As shown in Figure 5 below, we can find two types of transformation relations among artifacts.

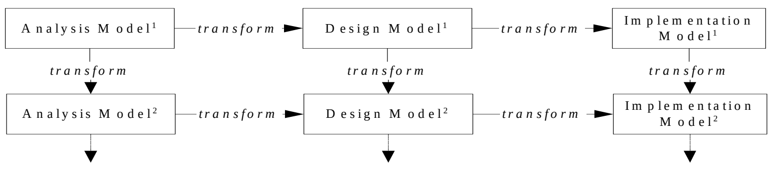

Figure 5. *Refinement of Transformation of Models*

This figure can be seen as a further refinement of Figure 3. The horizontal direction of the transformation in Figure 5 is for making artifacts more concrete and its transformation goes along the progress of software development from an analysis model (requirements specification) to an implementation model (program), while the vertical direction holds the same abstraction level and indicates only the refinement of the same model.

In the horizontal transformation, it is important to preserve the quality from the analysis model^1 to the design model^1, and finally to the implementation model^1. This preservation of quality can be supported by automating the preservation of transformation links, the storing of the various artifacts and the active rule support in producing and transforming artifacts.

The vertical transformation denotes model transformations of the same model. The reason for transformation may be due to introduction of evolutionary requirements or the need for a different representation of the same model. For example, in Figure 5 *Analysis Model^1* may be written in a natural language but transformed into *Analysis Model^2* written in a formal language to reason and ensure the quality of the analysis model. Any inconsistencies in the original analysis model can then be easily detected and corrected. This may require bi-directional traceability of the artifacts. In the same sense, *Analysis Model^2* may represent the analysis model with additional requirement. The updating of the artifacts may have direct impact on the subsequent models and require the retriggering of the transformation process. Automated support may be helpful to guide this process.

Meta-Models

Like conventional modeling, meta-modeling by its own can be a means to formalize different aspects of the software development process in order to support its automation. Each meta-model has its own focus and scope and solves a particular problem. Meta-models can be utilized as conceptual schemas for repositories that hold knowledge on artifact production and manipulation. Meta-models may be defined for artifact types, like in the UML, but also for heuristic rule support or process support. Meta-modeling has basically focused on modeling artifact types, however, for an active support of software development methods it is required that also meta-models are generated for coping with heuristic rules and process support.

Meta-models may also be needed to couple different CASE tools and to provide interoperability. Since CASE tools may be based on different meta-models this results in the composability problem of meta-models. Current techniques for solving this issue is by providing Meta-CASE tools in which meta-models can be adjusted to the support the automation of different methods. Nevertheless, even then a change of the methods that are modeled might require the meta-models to change as well, and it may not be so easy to define an appropriate meta-model.

7. Conclusion

In this paper we have described the results of the workshop on automating methods. We have first presented the background on the notion of methods and identified that every method basically includes a process, rules and artifact types to produce artifacts. We have defined the rationale for applying methods and automating methods. It appears that automating methods requires knowledge on the software development methods, meta-modeling, method engineering techniques and knowledge on CASE tool development. We have explained the methodological framework for software development in Figure 1 and showed our focus of interest on defining CASE tools for developing and managing methods. In Figure 2 we have explained the four-level architecture of meta-modeling and refined this over the whole paper. Figure 3 has shown the various aspects of meta-models within one layer of the four-layered architecture. Hereby, software development is seen as a transformation of models, that might be themselves reflected on using meta-models to provide automated support. This observation highlighted several problems in automation of methods. Basically, we can define meta-models for artifact types, heuristic rules and the process.

We have introduced the quality factor of *automatability*, which refers to the possibility of automation for the corresponding methods. As a matter of fact some methods have a higher automatability degree than other methods. Nevertheless, automation might also be useful for flexible methods to support the human intensive but less conceptual activities.

8. Acknowledgements

It took us 302 e-mail communications to organize the workshop and finish this summary report. We would like to thank Hervé Albin-Amiot, Omar Aldawud, Christian Wege, and Toshiaki Aoki for their contribution to the discussion section in this report. The following persons attended the workshop:

1. Aksit, Mehmet, University of Twente.
2. Albin-Amiot, Hervé, Ecole des Mines de Nantes.
3. Aldawud, Omar, Lucent Technologies

4. Aoki, Toshiaki, JAIST
5. van den Broek, Pim, University of Twente
6. Dao, Michel, France Télécom R&D
7. Drori, Offer, Hebrew University of Jerusalem
8. Elrad, Tzilla , University of Chicago
9. Fuentes, Lidia, University of Malaga
10. Glandrup, Maurice, University of Twente
11. Hillenbrand, Gisela, Interactive Objects Software
12. Kobayashi, Takashi, Tokyo Institute of Technology
13. Libourel, Therese, LIRMM
14. Mughal, Khalid, University of Bergen
15. Van Paesschen, Ellen, Vrije Universiteit Brussel
16. Pons, Claudia, LIFIA
17. Revault, Nicolas, Univ. Cergy-Pontoise - LIP6
18. Roume, Cyril, LIRMM
19. Saeki, Motoshi, Tokyo Institute of Technology
20. Said, Jamal,K.U.Leuven
21. Sunyé, Gerson, IRISA
22. Thierry, Eric, LIRMM
23. Tekinerdogan, Bedir, University of Twente
24. Vallecillo, Antonio, University of Málaga
25. Wege, Chris, University of Tübingen
26. Yann-Gaël Guéhéneuc, Ecole des Mines de Nantes

The organizers would like to thank all the attendants for their participation to the workshop and the fruitful discussions that we had. Finally we would like to thank Manabu Ishida for his great efforts in converting this document from MS-Word to Latex.

9. References

1. Adelson, B, & Soloway E: *The role of domain experience in software design*. *IEEE Trans. Software Engineering*, SE-11(11), 1351-9, 1985.
2. Arrango, G: *Domain Analysis Methods*, in: Software Engineering Reusability, R. Schafer, R. Prieto-Diaz, & M. Matsumoto (eds.), Ellis Horwood, 1994.
3. Beck, K: *Extreme Programming Explained,* Addison-Wesley, 2000.
4. Berg, van den K: *Modeling Software Processes and Artifacts*, In Bosch, J. and Mitchell, S. (Eds), *ECOOP'97 Workshop Reader*, LNCS 1357, Springer-Verlag, pp. 285-288, 1997.
5. Bezivin, J: *Who is afraid of ontologies,* OOPSlA 1996 Workshop on Model Engineering, Methods, and Tool Integration with CDIF, 1998.
6. Booch, G., Rumbaugh, J., & Jacobson, I: *The Unified Modeling Language User Guide*, Addison-Wesley, 1999.
7. Budgen, D: *Software Design*, Addison-Wesley, 1994.

8. Chikofsky, E.J; *Computer-Aided Software Engineering (CASE)*. Washington, D.C. IEEE Computer Society, 1989.
9. Cockburn, A: *Agile Software Development*, Addison-Wesley Longman, 2001.
10. Feiler, P.H., & Humphrey, W,S: *Software Process Development and Enactment: Concepts and Definitions*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1991.
11. Gane, C: *Computer-Aided Software Engineering: The Methodologies, the Products, and the Future,* Englewood Cliffs, NJ: Prentice Hall, 1990.
12. Ghezzi, C., Jazayeri, M., & Mandrioli, D: *Fundamentals of Software Engineering*. Prentice-Hall, 1991.
13. Humphrey, W.S: *Managing the Software Process*, Addison-Wesley, 1989.
14. Jacobson, I., Booch, G., & Rumbaugh, J: *The Unified Software Development Process*, Addison-Wesley, 1999.
15. Johnson, R., & Foote, B: *Designing Reusable Classes.* Journal of Object-Oriented Programming. Vol. 1, No. 2, pp. 22-35, 1988.
16. Kruchten, P.: *The Rational Unfied Process: An Introduction*, Addison-Wesley, 2000.
17. Pressman, R.S. *Software Engineering: A practitioner's approach*, Mc-Graw-Hill, 1994.
18. Riel, A*: Object Oriented Design Heuristics*, Addison-Wesley, 1996.
19. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., & Lorensen, W: *Object-Oriented Modeling and Design*, Prentice-Hall, 1991.
20. Rumbaugh, J., Jacobson, I., & Booch, G. *The Unified Modeling Language Reference Manual*, Addision-Wesley, 1998.
21. Saeki, M: *Method Engineering*. in: P. Navrat and H. Ueno (Eds.), Knowledge-Based Software Engineering, IOS Press, 1998.
22. Tekinerdogan, B. & Aksit, M: *Adaptability in object-oriented software development: Workshop report*,in M. Muhlhauser (ed), Special issues in Object-Oriented Programming, Dpunkt, Heidelberg, 1998.
23. Tekinerdogan, B., & Aksit, M: *Providing automatic support for heuristic rules of methods*. In: Demeyer, S., & Bosch, J. (eds.), Object-Oriented Technology, ECOOP '98 Workshop Reader, LNCS 1543, Springer-Verlag, pp. 496-499, 1999.
24. Visser, W., & Hoc, J.M: *Expert software design strategies*. In: Psychology of Programming, Hoc, J.M., Green, T.R.G., Samurçay, R, & Gilmore, D.J. (eds). Academic Press, 1990.