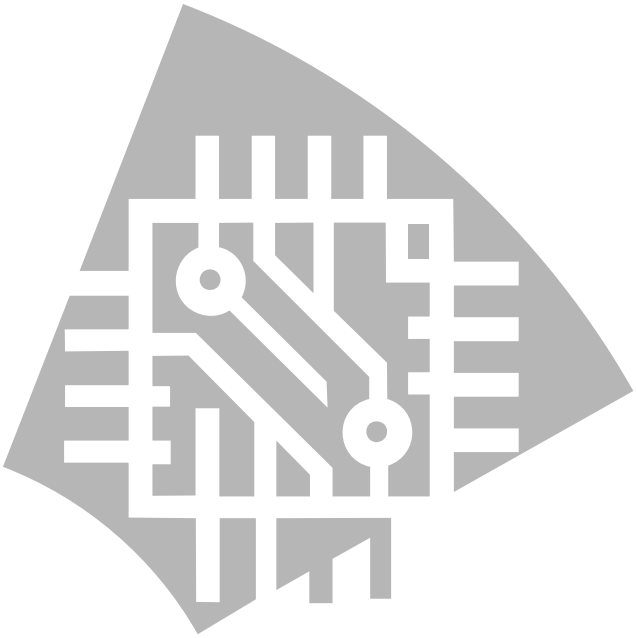
MSc Thesis of Sander Hartogensis
afstudeernummer: 540

Improving Software Development

An evaluation of tool and process related inefficiency in software development



'The only realistic way to increase software quality and productivity significantly is through automation.'
Carma McClure, CASE is software automation, 1989

'A tool is only a means'

Ben Flokstra, 2003

Thanks

I would like to thank a number of people who have helped me in various ways creating this thesis.

First and foremost I thank my fiancée Jennifer Pfundt. Creating this thesis was difficult for me, and without her love, support and help this thesis would not be here.

I thank the people who kindly spent the time and effort for interviews. Their input has largely shaped this thesis. Their names are Hans Bossenbroek, Jan-Willem Hubbers, Ben Flokstra, Peter van der Molen, Maarten Steen, Marc Lankhorst, Rob Vens and Helgo Broxterman.

Finally I thank my supervisor Erik Proper, whose valuable insights have been indispensable.

Text conventions

Whenever I use he, him or his in this thesis I could also have used she, her or hers.

Quotes of external sources are printed in cursive letters within quotation marks. If a term in this thesis is printed in bold letters, it means it is being defined. If a word appears in bold letters within a quote, it means it is in a definition from someone else but me that I adopt.

Abstract

Software development in general is not performed as efficiently as possible. The causes for this can be, among others: Lack of insight in the relationships between artefacts associated with a software development project; too much effort in adapting a development method to a specific situation; too great complexity of software; too much repetition in software development; possibly a lack of automation in currently available tools.

My analysis shows that most of the basic human competencies can not be had by a computer. By mapping these competencies on the actions of which the software development process consists, I have shown that most of the actions of the software development process consists can not be fully automated.

I have researched ways of remedying each of the causes for inefficiency that I have mentioned. But I have research two possible remedies more elaborately. These are Model Driven Architecture and method engineering.

My conclusion, after analysing the kinds of inefficiencies, the possible solutions, and the wishes and predictions of software developers, is that software development can be improved by pursuing the following developments: Mature component based development; advanced web services; advanced method engineering; more complete standardised project repositories.

Preface

In this chapter I explain what this thesis is about and how I came by the idea to do this project. This is a thesis about improving software development, by improving software development tools and development processes, with an emphasis on the tools.

Initially I chose Model Driven Architecture (MDA) as the subject, after Gary Hass, who worked at CIBIT in Utrecht, and I talked about having me do research on that subject at CIBIT. MDA is a theoretical framework of ideas about software development, and it is developed by an organisation called the Object Management Group (OMG).

That idea did not work out because I came to the conclusion that I could not find enough consistent and independent information on MDA.

But researching MDA had inspired me, because at the time I believed it carries the promise of new kinds of tools that support software development in a new way, and I wanted to do something with it. I decided to make a thesis about tool-support for the software development process in general, and to include a chapter about MDA. That way I could research MDA without having to worry if I could come up with enough material for a whole thesis. And I decided that looking at other ways than MDA to improve the software development process with tools would broaden my perspective.

While I was writing this thesis, I noticed that some of the concepts I was researching had not only tool aspects, but also process aspects. Then I decided to let this reflect in the research question of this thesis.

1 Introduction

This chapter describes the problem area, problem, solutions, research goal and research method of this thesis. The sections 1.1 to 1.3 together describe the problem area of the problem. The problem area is basically the area of software development. Section 1.4 describes the problem, which, in short, is that software development is often not performed as efficient as possible. I have tried to solve this problem by formulating and answering a research question. This question is formulated in section 1.5, along with six research sub questions. Section 1.6 describes how this thesis answers the main research question and the research sub questions, and explains the structure of this document. Section 1.7 explains why this thesis is relevant, and to whom.

1.1 The different natures of software development

Software development is a very broad subject area, because many different kinds of software exist. Examples of kinds of software development are the following:

- embedded software - for example in phones and pacemakers
- desktop software - for example word processors
- operating systems
- software to support business processes in organisations.

Developing these different kinds of software requires different kinds of software development processes, although there are similarities, as there are between manufacturing processes in general. Each manufacturing process has, in some order, the following activities:

- definition - finding out *what* the product should do
- modelling - defining *how* the product should do it
- construction - realising the model in something that works
- deployment - bringing about the situation where the users can use the product

As I am most familiar with the kind of software that supports business processes in organisations, I choose focus on improving the development of that kind of software in this thesis. Nevertheless, a lot of the theory from this thesis

is applicable on the development of other kinds of software.

1.2 Short History of software development

In this section I briefly describe the trends in software development since the late 1960s, with an emphasis on the role of automation in software development. I have included this section to put software development, as it is currently done, into context. The information in this section is taken largely from [McClure89] and [Whittaker03].

In the 1950s software engineering was a small field of work. *'Computers existed to solve highly mathematical problems, and their first programmers were mostly the people who defined and derived the equations.'* [Whittaker03] Computers and programming environments were only capable of executing basic tasks and were not complicated [Whittaker03].

In the 1960s, when computers became more accessible and powerful, programming environments became more powerful and easier to apply to problems. As a result technically less-skilled people became able to develop software. Also, the problems for which software was written became more complex [Whittaker03]. So, the problems became more complex while the software developers became less qualified.

In the 1970s there was a crisis in software development. This had three causes. Firstly, the development of problems becoming more and technically less-skilled software developers continued. Secondly, improving software development technology demanded a less rigorous way of working, so programmers could program in a more ad hoc fashion. Software developers often had access to a software compiler of their own, so they did not have to wait for a chance to compile their program on a central computer. Compilation was available any time a programmer wanted a quick syntax check. Thirdly, software was used to solve new problems. Software was not only used for mathematical algorithms, but also to help companies do their business faster and more efficient [Whittaker03]. The demand for software grew enormously. In short, there was a situation where technically less skilled people were making more software, of new kinds, and of increasing complexity, in a more ad hoc fashion than before. Carma McClure described software development from that time as a *'private art, left to the whim of individual programmers'* [McClure89].

The solution was offered in structured software development methods, which

made software development more of an engineering-like disciplined process.

In the 1980s CASE (Computer Aided Software Engineering) tools became popular. These were aimed at supporting software modelling. At first they were only programs for drawing diagrams from a certain modelling method. Later on they became tools for keeping a repository of diagrams, and checking consistency between diagrams. Also in the 1980s IDE's (Integrated Development Environments) became more popular. [McClure89] An IDE is a program in which several activities which have to do with programming are integrated. Examples of common activities that are supported by IDE's are code editing, code compiling, linking object files, file management and debugging. At the end of the eighties a trend came into being to standardise the processes that governed software development [Whittaker03].

In the 1990s standardising and improving software development processes became more important. The Capability Maturity Model [CMM] was at the centre of this development [Whittaker03]. According to [Whittaker03] the central idea of the CMM is: '*Software development is a management problem to which you can apply proper procedures for managing data, processes, and practices to good end. Controlling the way software is produced ensures better software.*' So, people began to see software development more as a process in which managers should be closely involved, instead of an activity performed by software developers, independent of the rest of an organisation.

It is interesting to observe the ideas software developers have had about how automation in software development would look in what was then the future. For example Carma McClure [McClure89] expected that generation of models and code would become far more advanced in the 1990s than it eventually turned out to be. She said that along with code and test cases, CASE-tools would automatically generate help screens and documentation for the user. Her faith in the development in tools is also reflected in the quote on the front of this thesis: '*The only realistic way to increase software quality and productivity significantly is through automation.*'

1.3 Why do we develop software?

This section briefly answers the question why software is developed, or, in other words, how an organisation comes to the decision to start a software development project. As I mentioned in section 0, I shall restrict myself to the area of software that supports business processes in organisations.

Not everybody has the same idea of how a software development process starts. And not everybody has the same idea of at what stage talking about a software development process has shifted into the starting of a software process. But, wherever you draw the line in time between *before* and *during* the software development process, there is always something before that line, that caused the software development process to start. Because the software development process is such an important subject in this thesis, it is useful to take a step back and look at how a software development process is initiated.

The management of an organisation can notice a discrepancy between the way an organisation should function and how it really functions. When that happens, the management of the organisation can decide to make a change in the organisation. Actions that bring about change in an organisation can vary greatly, because there are many elements that can be influenced by the management. Examples of actions to cause change in an organisations are: educating personnel, organising team-building workshops, changing business processes, changing manufacturing processes, redesigning the workspace, changing the management structure, and changing the software that is used to support the organisation.

If the management of an organisation decides there is a discrepancy between the way the organisation should function and the way it really functions, it can make an analysis of the situation. If that analysis indicates that the solution to the problem is a change in the software that the organisation uses, the organisation should produce a set of preconditions for a software development process. These preconditions can include general requirements of the software, requirements of the project itself, schedule-restrictions and resource restrictions.

1.4 Problem

People have always looked for ways to reduce the resource-cost of software development and to increase the quality of the produced software. If software development is done inefficiently, then this is a problem, because the cost of software development would be unnecessary high.

Types of inefficiency can be categorised in various ways. The categorisation that I choose is the following:

1. People related inefficiency
2. Tool related inefficiency

3. Process related inefficiency

Here are the definitions for inefficiency and the kinds of inefficiency. All definitions are in the context of software development:

Inefficiency is waste of resources.

People related inefficiency is inefficiency that can be remedied by influencing workers. People related inefficiency can be the result of for example bad management of human resources or motivation problems.

Tool related inefficiency is inefficiency that can be remedied by better tool support.

Process related inefficiency is inefficiency that can be remedied by improving the processes that drive the software development.

In chapter 3 I explain that in general tool related inefficiency and process related inefficiency are unnecessary high in the software development process.

1.5 Research question

The goal of this thesis is to help solve the problem formulated in section 1.4 by answering the following question, which is the main research question:

How can improvement of tool support and development processes help make software development more efficient?

The sub questions which help answer the main question are:

1. What exactly is software development, and how is it done?
2. Which types of inefficiency can be observed in a software development process, and how can these inefficiencies be remedied?
3. Which actions of the software development process can be automated?
4. What degree of automation of the actions from the software development process is offered by existing tools?
5. What wishes for improvement in software development are expressed by software developers?
6. What new developments can possibly lead to a more efficient development process in the future?

Questions 1 and 3 are questions that require the creation of a theoretic frame-

work. The other questions are questions that require making observations in the real world. I relate the theoretic framework to the observations of the real world.

In this thesis I try to answer each of these sub questions. In the chapter Conclusion, I answer the main research question.

1.6 Research method

The research consisted of interviews with several software developers and literature research. In this section I explain the role of the interviews in this research, and what research I did to answer the research questions.

The role of the interviews

The people I interviewed are: Hans Bossenbroek, Jan-Willem Hubbers, Ben Flokstra, Peter van der Molen, Maarten Steen, Marc Lankhorst, Rob Vens and Helgo Broxterman. All of these people are software development consultants or programmers, and consider them all to be software developers. Bossenbroek Hubbers, Flokstra and Van der Molen are contacts of my supervisor. Steen, Vens and Broxterman are people I have met before. Steen redirected me to Lankhorst after I had interviewed him. Bossenbroek is a consultant at Luminis, a software consultancy company in Apeldoorn. Hubbers is a consultant who works at Ordina Public Consulting, a software consultancy company in Rosmalen. Van der Molen is a software architect at BCICT, the IT division of the dutch tax authorities, in Apeldoorn. Steen and Lankhorst work at the Telematica Instituut, a research institute in Enschede. Steen works there as a scientific researcher, Lankhorst is head of the group of software engineers of the Telematica Insitute. Vens is a consultant who works at CIBIT, a consultancy company in Utrecht. Broxterman is a programmer who works at 3Plan in Veenendaal.

The main purpose of these interviews is to find out about the way they develop software and how they think that the efficiency of software development could increase.

I used a loose method of interviewing and did not stick closely to the questions, as long as the topic of the conversation was software development. By interviewing this way I could also find out what topics are relevant to software development that I did not know about.

The answering of the research sub questions

The first research sub question, 'What exactly is software development, and how is it done?' I answer using a synthesis of my own knowledge and literature. This question is answered in chapter 2, 'A theoretical framework for software development.'

I answer he second research sub question, 'Which types of inefficiency can be observed in a software development process, and how can these inefficiencies be remedied?' using results from the interviews and from a synthesis from literature and my own knowledge. This question is answered in chapter 3, 'Types of inefficiency'.

The third research sub question, 'Which actions of the software development process can be automated?' is answered in chapter 4, 'What actions can be automated,' using a synthesis from my own knowledge and literature. A list of the parts of the software development process is taken from chapter 2, which contains a framework of concepts that underlie software developments.

The fourth research sub question, 'What degree of automation of the actions from the software development process is offered by existing tools?' is answered in chapter 5. This chapter relates the potential for automation from each action from the software development process, from chapter 4, to existing tools.

I obtained the answers to the fifth research sub question, 'What wishes for improvement in software development are expressed by software developers?' exclusively from the interviews. The chapter in which I answer this question is chapter 6, 'Wishes of software developers'. I elaborate on two wishes, Model Driven Architecture and method engineering, in chapters 8 and 9.

I answer the sixth research sub question, 'What new developments can possibly lead to a more efficient development process in the future?' in chapter 7. Chapter 7 gives an overview of new developments that came up during the interviews. Two developments from this chapter, Model Driven Architecture and Method Engineering have a double function in this thesis, because they are both wishes of software developers, and new developments that can possible lead to a more efficient development process in the future. These developments are discussed in chapter 8 and chapter 9.

I have found the information about Model Driven Architecture through literature research. I have found the information on Method Engineering through

literature research and one of the interviews.

1.7 Relevance

This thesis is primarily relevant to the software development industry, because it researches ways to make software development more efficient.

This thesis is also relevant to the academic community. The theoretical chapters of this thesis offer a deeper understanding of inefficiency in software development, software development itself, and the potential for automation in software development.

2 A theoretical framework for software development

2.1 Introduction

This chapter answers the first research sub question 'What is software development, and how is software development done'. This chapter presents a definition of software development, and a framework which describes a generic software development process. Such a framework is necessary in a thesis about software development, because it allows all theories about software development in this thesis to be related to a common body of concepts. All of the terms that are defined in this chapter are existing terms, which often have multiple meanings. With the definitions in this chapter I want to make sure that everybody who reads this paper can interpret these terms in the same way.

I have strived to make the definitions in this chapter describe the essence of the concept they define, and make them no more specific than that. I have also strived to make definitions that are not too generic to be unusable, but generic enough to be broadly applicable.

This thesis does not describe a specific method of software development. This is because I believe a specific method of development has to be selected for each software development project individually.

I do not intend to let the framework defined in this chapter describe all concepts that exist in software development. It only describes the concepts that need to be defined in order to correctly understand this thesis.

2.2 What is software development

When discussing software development in depth it is necessary to specify what exactly software development means. Of course, software development is 'making software', but not everybody has the same idea of what that means.

In my experience people regard software engineering and software development as the same thing. I will also regard these terms as synonyms, and in my search for a definition for the term 'software development' I will also consider definitions for the term 'software engineering'.

On [SEIDefinitions] a list of definitions of the related term 'software engineering' are given. A few examples of definitions are:

'[Software Engineering is] the establishment and use of sound engineering principles

(methods) in order to obtain economically software that is reliable and works on real machines.' [Bauer72]

'Software engineering is the technological and managerial discipline concerned with systematic production and maintenance of software products that are developed and modified on time and within cost estimates.' [Fairley85]

'Software engineering is an engineering discipline which is concerned with all aspects of software production from the early stages of system specification through to maintaining the system after it has gone into use.' [Sommerville00] (This one is not referenced on [SEIDefinitions])

There are problems with each of these definitions. These are the following:

The definitions of Bauer and Fairley lack neutrality. They do not just specify what software engineering is, but also how it should be done properly. They say that it should result in software that is 'reliable' and that it should be done 'on time and within cost estimates'. They apparently do not believe that bad software engineering is still software engineering. The definition of Sommerville does not have this problem, although he defines what software engineering *is concerned with* and not what it *is*.

I will try to give a definition of software development that does not describe of what activities it consists, but one that describes the essence of software development, independent of how the software development is executed. In short, in this section I want to give a definition that describes *what* software development is, and not *how* it is done (how it is done is described in chapter 2).

The definition I will use is:

Software development is performing activities which contribute to defining, modelling, constructing, verifying and deploying a software system.

This definition uses the terms defining, modelling, constructing, deploying and verifying. Each of these terms needs to be defined in order to understand the definition of software development.

Defining is the activity of making a list of requirements of the software system (*what* the system should do).

Modelling is the activity of defining *how* the software system works.

Constructing is the activity of making the software system itself.

Verification is the activity of verifying that the software system works as speci-

fied in the activity Defining.

Deploying is the activity of making the software system work on the specific computers the users wants it to be run on.

These definitions could also be made more specific, but then I would not just be explaining *what* software development is, but also *how* it should be done. I have chosen this definition because it basically is a blueprint of any engineering process, made specific for the area of software. Any engineering activity contains the activities of specifying *what* is to be engineered, explaining *how* it will be engineered, engineering the product itself, making sure it works (verification), and making it ready for use by a user.

2.3 Activities in software development

In the previous section I defined *what* software development is. In this section I describe *how* software development is done. I do this by presenting a framework of the activities that I consider to be the core activities of software development.

I have chosen to model an activity as something which can consist of other activities, as illustrated in figure 1.

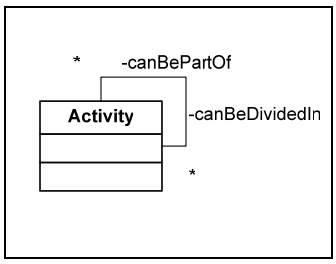


figure 1: UML class diagram. An activity can be subdivided in other activities.

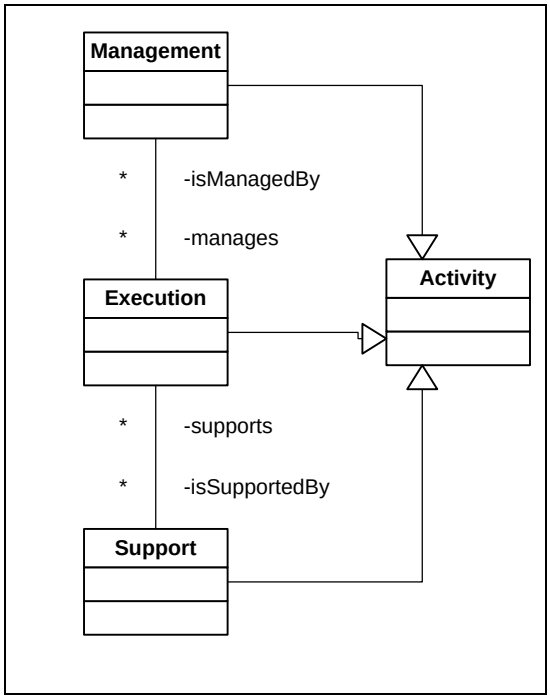


figure 2: The execution activities in a software development project are managed and supported.

The three main activities are execution, management and support. I have chosen for this first distinction between activities because their natures are very different from each other.

Execution is the activity of really making the product. Execution is the most essential activity, because it is this activity that directly yields a software product.

Management is the activity of regulating and controlling execution (this is a different kind of management than the management of an organisation).

Support is the activity of making execution and management easier. It would have been a legitimate choice to make management and support one single activity, because management only exists to make sure that the execution activity is performed well. So, in a way, management is a supporting activity. I choose to make the distinction between management and support anyway, because I want to make a distinction between the kind of support that controls (the management activity) and the kind of support that just facilitates (the support activity).

The rest of this section explains of what activities the activities execution, management and support can consist. I say 'can consist' instead of just 'consist', be-

cause no two software development projects are alike, and some may need these activities, some may not, and some may need other activities.

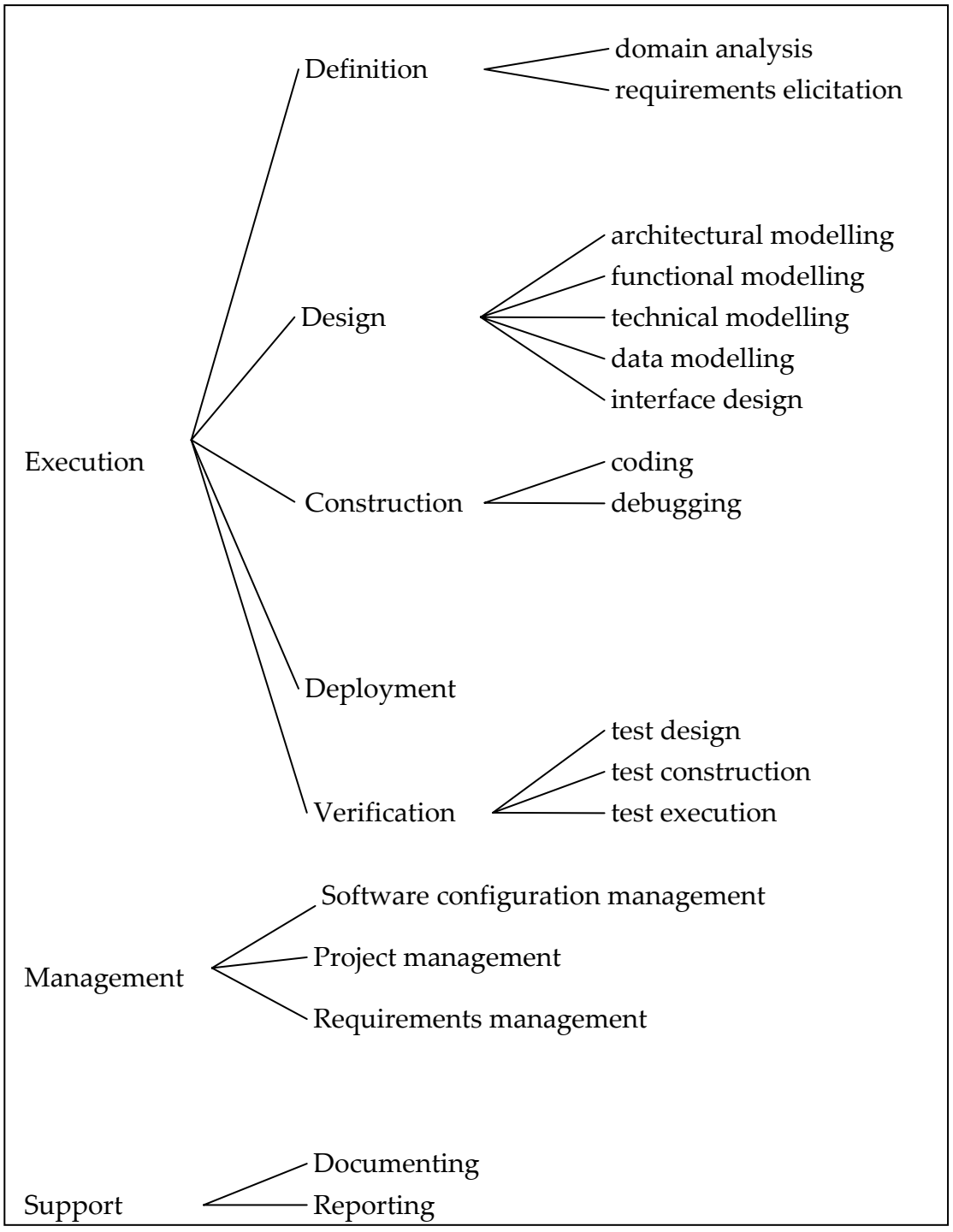


figure 3: An overview of activities in a software development process

The activity Execution is subdivided in the activities Definition, Modelling, Construction, Verification and Deployment. According to my definition of

software development in section 2.2, these are the activities that are essential to the software development process. These activities are defined in section 2.2.

The activities of which the activities Definition, Modelling, Construction, Verification and Deployment are composed, as shown in figure 3, are not essential to the definition of software development, I have chosen them as activities that are executed in a typical software development process. Here follows a list of definitions of these activities.

Activities that can be part of Definition:

Domain analysis is the activity of analysing the real-world context in which the software product is intended to run. The result of this activity is the **domain model.**

Requirements elicitation is the activity of determining the requirements of the software product. The result of this activity is the **requirements specification**.

Activities that can be part of Modelling:

Architectural modelling is the modelling of the architecture of which software applications will be a part of. The standard IEEE 1471 describes an **architecture** as: *'The fundamental organisation of a system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution.'* In practice an architecture will often be the collection of rules that were made, to which every piece of software in an organisation must adhere. For example: 'All software is written in C++', 'These XML schemas must be used', 'All shared services are accessible through SOAP'.

Functional modelling is the activity of identifying the components of which the software product is to be composed of, and defining the outward visible behaviour of these components. The result of this activity is the **functional model**.

Technical modelling is the activity of defining how the software system implements the functional model. The result of this activity is the **technical model**.

Data modelling is the activity of modelling and structuring the data that is processed by the software product. The result of this activity is the **data model**.

Interface modelling is the activity of modelling the screens through which a person can interact with a software system. The result of this activity is a collection of pictures of screens, called the **interface modelling**.

Activities that can be part of Construction:

Coding is the activity of making the source code of the software product. The result of this activity is code that can be executed in some way, called **program code**.

Debugging is the activity of finding and removing errors from program code.

Activities that can be part of Verification:

Test modelling is the activity of modelling tests for the software product. The result of this activity is the **test model**.

Test construction is the activity of constructing the tests and making them ready for execution. The result of this activity is code that can be executed in some way, called **tests**.

Test execution is the activity of executing tests. The result of this activity is a collection of information about the tested software system, called **test reports**.

Activities that can be part of Management:

Software configuration management is the management of work products. A work product can be for example a source file, a model, a compiled program, a software test, or a complete release of a software product. In this context, management can be version management, sharing management, or the regulating of changes that are made to work products.

Project management is the activity of making sure that the project is successful, by managing people and resources.

Requirements management is the activity of managing changes in the set of requirements, and keeping track of which requirements are met by the software product under development.

Activities that can be part of Support:

Documenting is the activity of making documents that help developers understand the system under construction.

Reporting is the activity of making documents that gives the persons involved in the creation of a system information about the progress of the development process.

2.4 Models

The term 'model' is an important term in this thesis, and therefore I discuss it in a separate section.

The term model can have different meanings in different contexts, even within the context of software engineering. I define a model as follows:

A **model** is a description or representation of a system, existing or not, that is made in a modelling language. This model captures the properties of a system that are essential to that system for a certain purpose.

The terms modelling language and system need explanation. A modelling language can be any language that is fit to describe a system in. This includes graphical languages such as UML. A system can be anything. Examples of systems are existing software systems, software systems that will be developed in the future, computers, computer networks, business processes and organisations.

2.5 Conclusion

In this chapter I have answered the first research sub question: 'What is software development, and how is software development done'. I have answered the first part of the question, 'What is software development', by comparing existing definitions of the term 'software development' and distilling a definition of my own.

I have answered the second part of the research question, 'How is software development done', by describing the concepts of which a software development process consists, and the message that a software development process can be assembled from these parts. These definitions also serve as a framework to which the theories in this thesis can be related, so they can be interpreted correctly.

3 Types of inefficiency in software development

This chapter answers the second research sub question, 'Which types of inefficiency can be observed in a software development process, and how can these inefficiencies be remedied?'

Inefficiency can have many different causes. I choose to divide these in three categories:

- People related inefficiency
- Tool related inefficiency
- Process related inefficiency

This thesis focuses on tool related inefficiency and process related inefficiency, and remedies to those kinds of inefficiency. The other category of inefficiency is people related inefficiency. In the interviews I conducted it appeared that seven of the eight respondents found that people related inefficiency has a far greater impact on the software development process than tool related inefficiency. Flokstra said about this: *'A person is still more important that the tools he uses. [..] A tool is only a means.'* Because they found people related inefficiency to be such an important subject I decided to include an overview of some important people related causes for inefficiency issues that came up in the interviews.

3.1 People related inefficiency

According to my respondents, most causes for inefficiency are people related. As Bossenbroek said: *'Bad software engineers are a bigger problem than bad tools.'* Here is an overview of a few problems that cause people related inefficiency.

Lack of clarity about the status of the project

Hubbers gave an example of this problem: Sometimes the decision to start a software development project is not made explicitly. It sometimes happens that while the decision to start a software development project has not yet been made, people already start doing a requirements analysis. Requirements analysis should be done after a software development project has started, not before the decision has been made to start the project. Of course it is necessary to decide on a higher level what the results of a software development project should be before it starts, but one should not try to make a list of all the requirements of a computer system before the project to build it has officially started. Hub-

bers: *'Then [when you start making requirements before officially starting the project] you are one step ahead. That is one of the causes for inefficiency. Surreptitiously, the project already starts when there still is no project. Then complicated discussions ensue about who is responsible for activities, and who you have to go to for a decision. [..] It sounds silly, but happens more often than you would think.'*

Setting too much project goals for one project cycle

Some software development methods, like RUP (for an introduction see [Kruchten00]]), use a cyclic approach to software development. This means that the project is divided in several project cycles, and that for each of these project cycles a cohesive set of project goals is set. According to Lankhorst a common cause for inefficiency is developers setting too much project goals for one project cycle.

Feature creep

According to Lankhorst often people who have commissioned the software product come up with increasingly many extra features for the software product under development, as the project progresses. This phenomenon is called feature creep. A common cause for inefficiency for project managers is to allow this, and keep adding those features as project goals.

Lack of attention for coherence of systems

According to Lankhorst, a common cause for inefficiency is developers focusing too much on one piece of software at a time, without considering the coherence with software systems that are already present in its environment. When this happens, the developers risk creating an incohesive jumble of systems, instead of a carefully planned IT-architecture.

Problems caused by people related inefficiency can result in projects failing or becoming more time- and money-consuming than expected. In the next section I will show that this is not the case with problems caused by tool related inefficiency.

3.2 Tool and process related inefficiency

When I asked the respondents of my interviews if they see tool related inefficiency as a problem, they all said no. Nevertheless, in the course of the interviews two respondents named two problems that can be classified as tool related inefficiency. Furthermore there is one possible cause for inefficiency that

that I perceived myself. All these causes for inefficiency are discussed in this chapter.

I choose to combine the discussion on tool and process related inefficiency in one section, because one of kinds of inefficiency can possibly be remedied by improving the combination of tools an processes.

Software developers having an unnecessary lack of insight in the relationships between artefacts associated with a software development project

Flokstra said that that often inefficiency is caused by lack of insight in the relationships between all artefacts associated with a software development project. Examples of artefacts are: models, descriptions of the entities of which a model exists, requirements, test descriptions and additional design documents. If one such element changes in the course of a project, it should be clear immediately which other elements are affected, but often this is not the case in a software development project.

A solution for this type of inefficiency is an advanced repository, which is a tool. This is discussed in section 6.1.

This is a kind of tool related inefficiency, because it can be remedied by better tool support. Remedying this kind of inefficiency would make the activities design and construction from the software development process more efficient.

Unnecessary effort in adapting a development method to a specific situation

Flokstra said that today an unnecessary amount of effort goes into the selecting and adapting of development methods for specific project situations. When a software development project starts, managers decide which software development method will be used, or which mix of software development methods. This means that the managers need to decide for example what kind of software will be used, what artefacts need to be delivered and how the project will be planned. Then this method or mix of methods often needs to be adjusted to the current situation. If this selecting and adapting could be completely or partly automated, it would cost less effort.

This type of inefficiency is also discussed in section 6.2. A remedy for this type of inefficiency is method engineering. This is discussed in chapter 9.

This is a kind inefficiency that is both process related and tool related, because it can be remedied by tools that improve processes. Remedying this kind of inefficiency would make the activities management from the software develop-

ment process more efficient.

Unnecessary complexity of software

This problem is process related, but may be remedied by a different style of software development. Today the task of managing the complexity of software systems that are expanded after their initial creation is a challenging one. Vens says that today especially business oriented applications are more complex, and thus more difficult to maintain, than necessary. Object orientation and components help encapsulate complexity, but Vens' own theory of a business centred architecture suggest that complexity can be reduced further than it already is in most organisations, thus making software development more efficient. This business centred architecture is briefly discussed in section 6.3.

This is a kind of process related inefficiency, because it can be remedied by influencing the way that software is developed. Remedying this kind of inefficiency would make the activities design and construction from the software development process more efficient.

Unnecessary repetition in software development

This problem is process related, but may have consequences for tools. Modelling of software is sometimes done by making several models in a row, each of which has more detail than the last. This leads to a practice where the modeller repetitively reformulates the model. One can wonder if a practice would be possible where the modeller does not have to reformulate each model from scratch, but can build his models incrementally, only adding information and never repeating it. I have formulated this problem myself.

Originally, I believed MDA offers a solution to this problem, after only having heard about MDA. After having discussed MDA with an experienced software developer, I believed that MDA enabled automatic model transformations. A developer could develop a model, independent of a technical platform, and then apply an automatic model transformation that transforms this model to a model with the same functionality, only made specific for a certain technical platform. That way, my idea could be realised of a practice where information from models never has to be reformulated.

However, after researching MDA, I concluded that MDA is only theoretical framework, for already existing concepts. This framework might help in finding a solution to this problem. But MDA does not offer a usable solution.

This is a kind of process related inefficiency, because it can be remedied by influencing the way that software is developed. Remedying this kind of inefficiency would make the activities design and construction from the software development process more efficient.

Possible lack of automation by tools

Possibly the level of automation in tools that are available today is not sufficient. In the chapter 4 I make an analysis of the potential for automation of the actions of the software development process, as defined in chapter 2. In chapter 5 I show that the tools that are available today do offer the automation that is theoretically possible according to chapter 4.

3.3 A comparison between types of inefficiency

Inefficiency caused by insufficient tool support is generally not a cause for project failures or unexpectedly high consumption of time or money. Usually software engineers know the tools that they use, and can make a reasonable estimate of how fast they can develop their software using those. These tools probably will not show unexpected behaviour during the project, so the tools are not likely to be the cause of unexpected project delays and higher costs. I suspect this is the reason that software developers do not see insufficient tool support as a problem; it does not stand out as a problem, while people related problems such as bad management can stand out much more spectacularly, as I have explained in the previous section.

3.4 Conclusion

In this chapter I answered the second research sub question: 'Which types of inefficiency can be observed in a software development process, and how can these inefficiencies be remedied?'

The types of inefficiency that can be observed in a software development process are people related inefficiency, tool related inefficiency and process related inefficiency.

The interviews revealed that software developers believe the most important causes for inefficiency are people related. However, people related causes for inefficiency are beyond the scope of this thesis.. In this chapter I have explained that inefficiency that appears to be process related can be also tool related.

In the introduction of this thesis I have explained that I only extensively discuss

tool and process related inefficiency. The tool and process related inefficiencies mentioned in this chapter are:

- Unnecessary lack of insight in the relationships between artefacts. This can be remedied by an advanced repository.
- Unnecessary effort in adapting a development method to a specific situation. This can be remedied by method engineering.
- Unnecessary complexity of software. This can be remedied by Vens' business centred architecture.
- Unnecessary repetition in software development. I have not found a remedy for this kind of inefficiency.
- Possible lack of automation by tools. Chapters 4 and 5 show that this is probably not a problem.

The reason that my respondents do not perceive tool related efficiency to be a big problem is the following: People and process related inefficiency usually causes unexpected increase in cost, and tool related inefficiency does not.

4 What actions can be automated

4.1 Introduction

This chapter answers the third research sub question, 'Which actions of the software development process can be automated?' This is an important question, because, if a part of software development be automated, it can possibly be done more efficiently.

A list of actions of which the software process consists is introduced in chapter 2, which contains a terminological framework for software development. In this chapter I describe a list of competencies which are relevant to the activities of which the software process consists, and I determine which of these competencies can be mastered by a computer. For each of the activities of which the software process consists I list what competencies are needed for executing it, and if it can be fully automated.

4.2 Competencies relevant to the software development process

Here follows a list of competencies that are relevant to the software development process, with definitions. The competencies Conceptualisation, Diagnostic use of concepts and Theory building are from [Boyatzis].

The word 'kind' is used several times in this chapter. A kind is like a type, but has a wider meaning. A non-predefined kind can not be understood by a computer.

Having knowledge means having certain data, paired with information on how to use these data. Having knowledge implies the ability to store and retrieve knowledge.

Understanding languages means being able to process information in a natural or formal language. Understanding languages implies the ability to interpret and produce language.

Recording data is the simple ability to receive and store data.

Creative thinking is the ability to create solutions of non-predefined kinds to problems.

Conceptualisation is the ability to create a system of concepts from an assortment of information.

Diagnostic use of concepts is the activity of identifying and recognising patterns in an assortment of information, matching concepts to these patterns and interpreting them through these concepts.

Social skills is the ability to understand the full spectrum of human communication, and to manipulate communication with a human using the full spectrum of human communication. The full spectrum of human communication involves not only speech and writing, but also understanding of etiquette, human character, meaning behind words, tone of voice and body language.

Rationalisation is the ability to make choices, based upon a rational explanation. It is the ability of logical thought. At every point in a thought process multiple choices can be made concerning the next step in that process. An agent capable of rational thought can make choice for a next step, and is able to give a rational explanation for that choice.

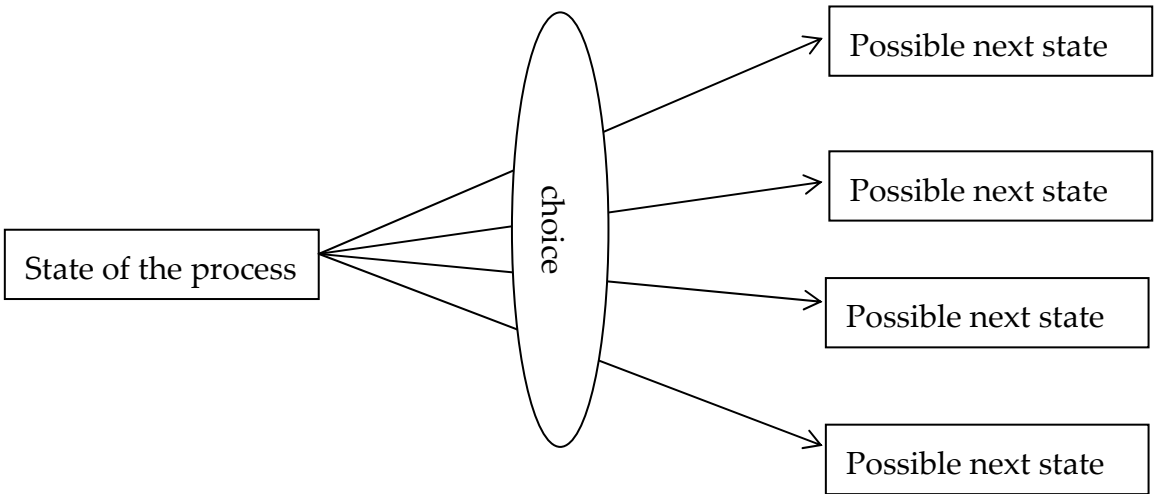


figure 4: In every state of the thought process a choice has to be made concerning the next step.

Communication is the ability to understand expressions from a language and to produce expressions from a language.

Theory building is the activity of creating a system of concepts and relationships between concepts which explains why an assortment of information is as it is, and how it will change under different circumstances.

4.3 What competencies can a computer have

This section lists which of the competencies relevant to the software development process (mentioned in section 4.2) can be had by a computer.

Before I discuss which of the competencies relevant to the software development process can be had by the computer, I will explain a basic assumption I make about the abilities of computers. I assume that a competency can only be had if a procedure can be written to implement it, that does not use techniques from the field of artificial intelligence.

competency	can be had by a computer
having knowledge	x
understanding languages	x
recording data	x
creative thinking	-
conceptualisation	~
diagnostic use of concepts	-
social skills	-
communication	x
Theory building	~
rationalisation	~

table 1: A list of competencies relevant to the software development process. An 'x' means a competency can be had by a computer, a '-' means it can not and a '~' means it can be had by a computer under limited circumstances.

In table 1 is a list of competencies relevant to software development. The table says of each competency if it can possibly be had by a computer, can not be had by a computer, or if it depends on the situation.

If a competency can possibly be had by a computer, a procedure can be written that enables a computer to have this competency. If a competency can not be had by a computer, no procedure can be written that enables a computer to have this competency.

Having knowledge – Knowledge is defined as data, paired with information on how to use the data. A computer can have knowledge, as long as the data and

the information on how to use it are of known types.

Understanding languages – A computer can parse a well defined language. When semantic data is associated with the various language elements, a computer can be said to have the ability to understand a language. When a computer produces output, procedures can be written to convert this to an expression that adheres to the semantic and syntactic rules of a well defined language.

Recording data – A computer can record data.

Creative thinking – No procedure can be written that produces data of a kind, that is not of a predefined kind. Therefore, a computer can not have the competency of creative thinking.

Conceptualisation – In a limited environment a computer is able to have this competency. The limits to the environment have to be that the information that has to be conceptualised is in a fixed kind, and that the kind of the concepts is also fixed. Information from the real world is not of a fixed kind, and useful concepts, including information about the meaning of the concepts, can not be expressed in a fixed kind.

Diagnostic use of concepts – This requires an understanding of concepts by the computer, which requires that any concept can be expressed in a fixed kind. This can not be done.

Social skills – This requires understanding of etiquette, human character, meaning behind words, tone of voice and body language. A computer can not have this understanding.

Rationalisation This is the ability of logical thought. Procedures can be written to implement logical thought, on the condition that the subject matter and the possible results of the thought process are in a fixed kind.

Communication This requires the ability to parse expressions from a language and to produce expressions from a language. Procedures can be written that implement this ability.

Theory building A computer can do this when the information about which the theories are to be built is in a fixed kind. It can not be done with information from the real world.

In conclusion, competencies that a computer can not be capable of, are competencies which require creative thinking, or the handling of non-predefined

kinds.

4.4 A scale of support for activities by tools

A tool does not have to fully automate an action to be useful, a tool can also just offer support to a user that performs the action himself. I propose a scale of support for activities by tools. The purpose of this scale in this chapter is to give insight into what tools can do for a software developer.

1. The activity is automatically performed by the tool.
2. The tool explicitly facilitates the performing of the activity by the user.
3. The tool does not explicitly facilitate the performing of the activity, but the user is able to perform the activity using the tool nevertheless.

For example, a simple text editor supports the activities debugging, coding, test construction, documenting and reporting to the third degree; all those activities can be performed using only a text editor, although a text editor is not especially created for this purpose. A tool that incorporates JUnit [JUnit], a test framework for Java code, supports the activities test construction and test execution to the second degree, because JUnit is created for just this purpose.

Level 3 support for an action is possible for every action in the software development process, because every action ca be supported by a text editor or drawing program.

The separation between level 2 and level 3 support is not strict. Take for example an arbitrary tool that supports UML modelling. It can be argued that this tool supports data modelling to the second degree, because data structures can be expressed in UML. It can also be argued that this tool supports data modelling to the third degree at best, because in many cases UML offers insufficient power of expression for data structures.

4.5 What activities can be automated

In table 2 is a list of the activities from the software development process (taken from chapter 2). This table says of each of the activities which competencies are needed to perform this activity. In table 2 I say in the rightmost column if an action can be automated or not. So I only say if it can have level 1 support by a tool (see section 4.4), or not. I have chosen not to say if it can have level 2 or level 3 support by a tool for two reasons. Firstly, as I have explained in section 4.4, it is not possible to make a strict separation between level 2 and level 3 sup-

port. Secondly, every action can at least be supported by level 3 support.

	have knowledge	understand languages	record data	creative thinking	conceptualisation	diagnostic use of concepts	social skills	communication	theory building	rationalisation	Can be fully automated
Definition:											
domain analysis (1)	x	x	x	x	xx	xx	-	x	x	x	-
requirements elicitation (2)	x	x	x	x	x	x	x	x	x	x	-
Modelling:											
architectural modelling (3)	x	x	x	xx	xx	xx	x	x	xx	x	-
functional modelling (4)	x	x	x	xx	xx	xx	-	x	xx	x	-
technical modelling (5)	x	x	x	xx	xx	xx	-	x	xx	x	-
data modelling (6)	x	x	x	xx	xx	xx	-	x	xx	x	-
interface modelling (7)	x	-	x	xx	xx	xx	xx	x	xx	x	-
Construction:											
Coding (8)	x	xx	x	x/-	x	x	-	x	x	x	x/-
debugging (9)	x	x	x	x	x	x	-	x	-	x	-
Deployment (10)	x	x	x	x	x	x	x	x	-	x	-
Verification:											
test modelling (11)	x	x	x	-	x	x	-	x	-	x	x

	have knowledge	understand languages	record data	creative thinking	conceptualisation	diagnostic use of concepts	social skills	communication	theory building	rationalisation	Can be fully automated
test construction (12)	x	x	x	-	-	-	-	x	-	x	x
testing (13)	x	x	x	-	-	-	-	x	-	-	x
Management:											
software configuration management (14)	x	-	x	-	-	-	x	x	-	x	-
project management (15)	x	x	x	xx	x	x	xx	x	-	x	-
requirements management (16)	x	x	x	x	-	x	xx	x	-	x	-
Support:											
documenting (17)	x	x	x	-	x/-	x/-	x	x	-	x	x/-
reporting (18)	x	x	x	x	x/-	x/-	-	x	-	x	x/-

table 2: The rows in this table represent the activities in a software development process, and columns represent competencies. Each cell tells if a competency is needed for performing an activity. An 'x' means yes, '-' means no, 'xx' means very much so, 'x/-' means 'it depends on the interpretation of the meaning of the activity.

Domain analysis can not be automated, because it involves conceptualisation and theory building based on information from the real world.

Requirements elicitation can not be automated mainly because it involves conceptualisation and theory building based on information from the real world. Also it involves extensive diagnostic use of concepts in relation to the real world, because the agent that performs this task has to apply its own knowl-

edge of software and software developments to the needs of the client. Communication with the client involves social skills.

Architectural modelling can not be automated mainly because it involves conceptualisation, creative thinking, theory building and diagnostic use of concepts using information from the real world. Conceptualisation has to be done of the available architectural options. Creative thinking has to be done to generate an architecture or a proposal for one. Theory building has to be done to predict which architecture will best fit the working situation. Diagnostic use of concepts has to be done to relate an architecture or a proposition for an architecture to the real world.

Architectural modelling, functional modelling, technical modelling, data modelling These activities have in common that a model is created, that in general can not be created by mechanically extending another model. This is demonstrated in the next paragraphs for each of these types of model.

An architectural model can not be mechanically created because it is not in some way an implementation of another model. The same goes for a data model. A functional model is in a way an implementation of another model, if the requirements for the system under development are considered a model. But these are generally formulated in natural language, in a way that does not allow automatic generation. A technical model is an implementation of a functional model, but can not be mechanically created because generally, when one knows what to do, one can not automatically determine how to do it.

In general, the creation of a model that can not be created by mechanically extending another model, the following holds: It can not be automated mainly because it involves conceptualisation, creative thinking, theory building and diagnostic use of concepts using information from the real world. Conceptualisation has to be done of the modelling options. Creative thinking has to be done to generate a model or a proposal for one. Theory building has to be done to predict which model will best fit the working situation. Diagnostic use of concepts has to be done to relate a model or a proposition for a model to the working situation.

Interface modelling can be automated if it is done by mechanically creating interface elements from every interaction that is described in a functional model. But if the interface has to be optimised for usability it can not be automated, it is a model that can not be mechanically created by extending another model. An

this kind of model can not be automatically generated for the reason described in the previous paragraph. The modelling of an interface that has to be optimised for usability can also not be automated because social skills are important for this activity.

Coding can be automated if the technical model is so detailed that coding can be done by mechanically extending the technical model. Otherwise it can not be automated for the previously described reason why models that can not be generated by mechanically extending another model can not be automatically generated.

Debugging can not generally be automated, because of its unpredictable nature. It involves finding and resolving of errors in the code with as only input the code itself, information about erroneous behaviour and error messages. The finding and resolving of errors involves especially theory building and creative thinking. A computer is not capable of that.

Test modelling, test construction and test execution can be automatically be done, if the functional model is formulated in enough detail. The functional model needs to describe for each function of the system the preconditions and postconditions. Then a testing tool can automatically create inputs that adhere to the preconditions, and test if the system generates output that adheres to the postconditions.

Software configuration management can be partly automated. The people management part of it, where tasks are divided among workers and everybody is told what to do involves extensive social skills, and can therefore not be automated. But the version management, sharing management and regulating changes can be automated. The configuration of a software configuration management system can not be automated, because it involves conceptualisation of the working situation, and theory building and diagnostic use of concepts with these concepts.

Project management can not be automated, because it relies heavy on social skills.

Requirements management can be partly automated. A system can be devised which keeps track of requirements which are already met and which have yet to be met. Also it could keep track of the relationships between requirements. People will have to tell the system what the requirements are, what their relationships are, and if they are met already.

Documenting can be automated under limited circumstances. If there is information recorded that may help developers better understand the element of the system that needs to be documented, it can be attached to that element. This information could be for example the requirement that this element helps satisfy. In general though, some amount of social skills is necessary to generate information that can help developers understand an element of the system under development.

Reporting can be automated, depending on the demands that are set on the documentation. The system that implements this can generate documentation using all the data that are related to a software development project to which it has access and of which it has understanding.

4.6 Conclusion

In this chapter I answered the third research sub question, 'Which actions of the software development process can be automated?'

It appears that very few activities can be fully automated. Most of the actions of the software development process require a competency that a computer can not have.

The activities that can be automated are coding, given that all the necessary creativity has gone into the technical model, and the testing related activities, given that the requirements are formal.

Competencies that a computer can not be capable of are competencies which require creative thinking, or the handling of non-predefined kinds.

Fortunately, automation can be used to support each of the activities of the software development process.

To find out how actions can best be supported by tools, it would be necessary to dissect the actions in sub-actions, and analyse each of these the way I analysed the main action from the software development process in section 4.4.

5 Automation in existing tools

5.1 Introduction

This chapter partly answers the fourth research sub question, 'What degree of automation of the actions from the software development process is offered by existing tools?' It is difficult if not impossible to give a complete answer to this question, because there are thousands of tools available.

My aim is to answer the question by giving an impression of what tools are available for each action of the software development process.

The original purpose of this chapter was to give information on a group of tools. In my research I encountered the problem that finding good information on tools is more difficult than I expected. I explain this problem in section 5.4.

5.2 What actions are supported in tools

This section contains a survey of the support for actions from the software development process (as defined in chapter 2) in tools today. For each of the actions I name a tool that supports the action, and I describe how this action is supported.

For most activities multiple kinds of tools exist that support different aspects of the activity. Also, for most activities multiple tools exist that support the same aspects. Because I only intend to give an impression of the tools that are available to support the different action from the software development process, I have chosen to only mention one tool per action.

For each tool I say what level of support it gives. The scale for levels of support is defined in section 4.4.

Domain analysis

Tool: MagicDraw [MagicDraw] (or any other UML modelling tool)

Support: Level 2 or 3

Description: MagicDraw supports UML modelling, and a the result of a domain analysis can be expressed in UML.

Requirements elicitation, Requirements Management

Tool: Rational Requisite Pro [ReqPro]

Support: Level 2

Description: Requisite Pro is a database system that is specifically aimed at storing and managing requirements. It allows users to do version management of requirements, and to define which requirements depend on each other.

Architectural modelling

Tool: Microsoft Office [Office] (or any other suite that supports text processing and drawing)

Support: Level 3

Description: Decisions about architecture can be of varying kinds. As I stated in the definition, it can for example be a description of coding standards, or of interfaces that have to be used. There is not specific tool support for architectural modelling other than the word processing and drawing programs of for example Microsoft Office.

Functional modelling

Tool: MagicDraw [MagicDraw] (or any other UML modelling tool)

Support: Level 2 or 3

Description: MagicDraw supports UML modelling, and the result of functional modelling can be expressed in UML.

Technical modelling

Tool: MagicDraw [MagicDraw]

Support: Level 2

Description: MagicDraw is specifically aimed at using UML for making technical models.

Data modelling

Tool: AllFusion ERwin Data Modeller [Erwin]

Support: Level 2

Description: ERWin Data Modeller is specifically aimed at supporting the designing of databases, and is therefore also specifically fit for the supporting of data modelling.

Interface modelling

Tool: Microsoft Visio

Support: Level 2

Description: Microsoft Visio is a drawing tool that has a template that facilitates the drawing of Microsoft Windows-style user interfaces.

Debugging

Tool: Visual Studio [VisualStudio] (or any other major integrated development environment)

Support: Level 2

Description: Visual Studio, like any other major integrated development environment, offers facilities for tracking what exactly happens during the execution of program code. Other tools exist for static analysis of program code.

Coding, Test modelling, Test construction, Test execution

Tool: Statemate

Support: Level 1

Description: Statemate is a tool that is aimed at embedded systems. It has its own modelling language and facilities for making models in that language. Statemate can run these models as simulations. It also has a language for formally expressing requirements. It can automatically generate tests that check if the system under development adheres to the requirements. These tests can then be run on a simulation of the system, or on the real system. Statemate can also generate complete working C code using a model.

Software configuration management

Tool: Rational Clear Case [ClearCase]

Support: Level 2

Description: Rational Clear Case can be used to store different versions of files, to manage access to files and to merge different versions of files.

Project management

Tool: Project InVision

Support: Level 2

Description: Project management is an activity that has so many subtasks that it is hard to define one kind of tool to support the activity. Project InVision is a

tool that supports the performing of several subtasks of project management, such as resource management, scheduling and keeping track of the status of multiple projects.

Documenting

Tool: Javadoc

Support: Level 2

Description: The purpose of Javadoc is to collect documentation that is interspersed in program code and to collect it in more conveniently arranged documents.

Reporting

Tool: SoDa [SoDA]

Support: Level 2

Description: SoDA is a documentation product that is part of several software development tools by Rational. It can gather information from Requisite Pro, the design tools from Rational, the test tools from Rational, and ClearCase. SoDA combines these kinds of information in project documentation it can generate.

5.3 Use in practise

The respondent of my interviews (see section 1.6) are all consultants or programmers. These people all use tools that have functionality that has been available to software developers for at least fifteen years. The tools that are used are mostly a tool for UML modelling (like MagicDraw [MagicDraw], Microsoft Visio [Visio]), a tool for coding support (for example VisualAge, Visual Studio or Borland Delphi) and Microsoft Word for various kinds of communication between project members.

5.4 Evaluating tools in practice

While researching software development tools on the internet, I noticed that it is very hard to get useful information about them. Because this is a problem that everybody who investigates software development tools will encounter, I feel a discussion of this topic is relevant in this chapter

The problem that I encountered while choosing tools to review, and performing those reviews, was that it is difficult to determine what a tool can actually do.

Most tool vendors publicise lists of the features that a tool has, but understanding these lists is often difficult if not impossible.

Given a feature and a tool, it can be difficult to determine if the feature is supported by the tool or not. An example of a feature of which it can be difficult to determine if a tool supports it is business modelling. There are products that specifically support business modelling, such as Aris Toolset [Aris]. But some kinds of business model can be created using the generic object oriented modelling language UML [UML], so any tool that allows a user to create UML diagrams can be said to support business modelling.

Many tools lack clear descriptions of their features. Several times during my research of tools I encountered feature lists that are very uninformative.

An example of a feature description that contains no useful information is in the document called 'ArcStyler 3.1 Feature List' that can be found on the ArcStyler site [Arcstyler]. A feature that this document lists is: *'Flexible verification of corporate IT standards based on mature industry standards.'* The document does not describe what this means and to what exact functionality in the tool this feature relates.

A feature can also seem more specific than this example and still be uninformative: 'System Modelling and delivery support' seems more specific, because most people will have an idea of what that is, but it still says little. What do the writers of the feature list mean by 'system modelling' and 'delivery'? What exactly does the tool do support these activities? A diligent researcher can try to find this out by reading technical manuals, but in the case of ArcStyler these are very detailed and not offer a clear overview of what the tool actually does.

A problem with feature listings is also that often features are listed that are nothing special but are formulated in a way as to suggest a very advanced feature. An example is of this is found in the document called 'Select Component Architect – A technical overview', from [Select]. It highlights *'some of the features that place Select Component Architect ahead of the competition.'* One of these features is *'Windows graphical user interface - Select Component Architect has familiar and contemporary user interface that helps users quickly adopt the tool so that its benefits may be instantly realised in software development projects.'* In short, a special feature of Select Component Architect is that it uses the Windows interface. This feature description expresses no useful information, and wading through this kind of information in search of useful information is a tedious job.

5.5 Conclusion

In this chapter I answered the fourth research sub question, 'What degree of automation of the actions from the software development process is offered by existing tools?'

The results of my survey are consistent with the predictions of chapter 4, where I concluded that no activity from the software development process can be automated, except for coding (in limited circumstances), and the test activities. These activities are fully automated in the tool Statemate. However, in order to automate these activities, the activities that produce the required input, have to be performed extra rigorously.

My findings were that comparing available tools costs far more effort than I have available, mostly because of poor documentation of the available tools.

A finding that surprised me was that while the number of tools keeps increasing, the respondents of my interviews mostly used the functionalities that have been available for a long time.

6 Wishes of software developers

This chapter answers the fifth research sub question, 'What wishes for improvement in software development are expressed by software developers?'

Three of the respondents expressed wishes for improvement in software development. The first wish relates to an improved repository for information relating to the software development project. The second wish relates to the formal development of software development methods. The third wish relates to MDA and a business centred architecture.

6.1 Advanced software development repository

Ben Flokstra expressed that he wishes that repositories would be more advanced and more standardised.

A **repository** is collection of the data that are relevant to a software development project.

Typically, a repository contains the models of the system under development, which are in a modelling language like UML. A repository could also contain other information that is relevant to the project, like, source code, requirements, use cases and unit tests.

Hubbers expressed the wish for a repository tool in which various links can be expressed between elements in the repository. Apart from links between information, such a tool would have to allow freedom of expression and should be as little restrictive as possible. Some examples of features are:

- Relating models or model elements to documents.
- A database of requirements, written in any form, which can be traced into the models and code.
- Generating reports about the status of the software development project, which are aimed at giving a stakeholder in the project customised information.

In the future repositories could be enhanced by standardising their format. This standardisation has already started; a UML model (in fact, a model in any modelling language that can be expressed in the Meta Object Facility [MOF]) can be encoded as an XMI file, which can be read by several tools already (for example [MagicDraw]).

The activities from the software development process that could be made more efficient with the help of an advanced repository are mostly design and construction.

6.2 Advanced method engineering

Many projects are not performed using LAD, RUP, DSDM or any other method straight from the book. These projects are adapted to the specific needs of the developers in a certain situation.

Ben Flokstra predicted that in the ideal world there would be a tool that, when given enough information about a software development project, can generate a complete method that could be instantly used to govern the project. This could contain among other items a list of tools that have to be used, instructions on what models to make and how to report. The generating of a method is called method engineering.

Chapter 9 elaborately discusses method engineering, and reviews Ben Flokstra's ideas about method engineering.

6.3 MDA in a business centred architecture

Rob Vens wishes that in the near future MDA will be accepted more broadly. MDA is important, said Vens in the interview I conducted with him, because it fits in with his vision for a business centred architecture.

Vens' definition for the term 'business centred architecture' is: *'An architecture that is designed with business models as basis. These models fully represent the business and adapt easily to the changes in the business, and even enable simulations of the business.'*

The concept from MDA that Vens considers to be important to his idea of a business centred architecture is that of the platform independent model (PIM, this is defined in section 8). Vens regards the business models from the business centred architecture to be PIMs, which should be used as a basis for the platform specific models (PSMs) of all applications that are developed in an organisation.

6.4 Conclusion

In this chapter I have tried to answer the fifth research sub question, 'What wishes for improvement in software development are expressed by software

developers?'.

I have given a list of avenues to possible improvements in tool support for the software development process. In short, these are:

- More complete standardised project repositories
- Advanced method engineering
- MDA in a business centred architecture

7 Possible future developments

This chapter answers the sixth research sub question, 'What new developments can possibly lead to a more efficient development process in the future?'

This chapter elaborates on five possible developments that can enhance software development in the future. These possible developments are based on predictions that were done by respondents in the interviews I have conducted. The developments are mature component based development, advanced web services, model driven architecture, method engineering and a yet unpredictable paradigm shift.

7.1 Mature component based development

Jan-Willem Hubbers expects that several decades in the future component based development will have become mature. This section explains what component based development is, what Hubbers explained it could be in the future, expands on Hubbers' ideas, and explains the immaturity of component based development today.

Mature component based development could make software development more efficient for the following reasons: Firstly, software development can be done quicker, because software development will rely more on predefined components. Secondly, software can be of better quality, if the components that are used are treated as complete products by the providers.

Component based development

This section explains what a component and component based development is.

Many different definitions of the term component are available. In this context a definition is needed that is specific to software development.

A **component** is a piece of software that has all of the following properties: it can be attached or removed from a software system, it encapsulates a coherent set of functionality, for some modelling purpose it is considered indivisible, and its interface adheres to a standard so that it can be added to any system that supports that standard.

Some parts of this definition need to be clarified or defined, which is done in this and the next two paragraphs.

The statement that a piece of software can be attached or removed from a soft-

ware system, means that it can easily be attached or removed from a software system, as a unit. The term 'easily' is subjective, which makes my definition of a component subjective.

The predicate 'for some modelling purpose indivisible' is best illustrated by an example: Imagine we are modelling a system that uses a separate database system with a standardised interface (for programmers: think of ODBC). For purposes of modelling our system, the database system is indivisible. That does not mean that the database system itself cannot be a component behind which a complicated data-warehouse is hidden that uses several other components to combine tables from databases located all over the world, written in ten different languages.

The **interface** of a piece of software is a part of that piece of software that can be used to communicate with that piece of software.

In this paragraph I formulate a definition of the term component based development. As with every computer-related term, many definitions exist. I try to find a definition that is not specific for one technique, but that is not so broad that it has little meaning. The obvious definition 'component based development is developing software or hardware using components' is too broad, because then almost every programming effort can be labelled component based development. This is because most function calls in a programming language can be explained as function calls to a component. I find the following definition made by myself useful:

Component based development is the practice of developing software by composing existing or newly made components into a software system, so that most of the functionality of the system is in a component.

A potential benefit of component development is that systems can be developed cheaper because components can be reused. Also, component based development, if done properly, can lead to a program code in which the structure of the model is clearly visible.

Mature component based development

Hubbers describes the practice of mature component based development as a practice of software development as follows:

Component repositories function as a kind of dictionary for available components. Software development is done using composition tools. A developer can

give a list of software system requirements to such a composition tool. Then this tool, which has access to the component repository, gives a list of components that could be used for building such a system. The software developer can compose these components into a working software system.

Maturity of component based development today

Hubbers says that software development today is not yet mature, and gives two reasons.

Firstly, it is difficult to let components communicate with each other if their interfaces are not compatible. There are tools and techniques to let all components communicate successfully with each other, but these are not widely accepted because most people are not able to use these.

Secondly, component based development is often practised badly. According to Hubbers, many practitioners of component based development make an error in thinking that component based development saves money and effort in the short term. This is not true because component based development is dependant on the components that are already available, and an organisation that has just started component based development has no components of its own yet, so there are no components to reuse.

A comparison with components in hardware

Hardware is mostly developed by combing existing component. According to Maarten Steen, in the future software development will be done more like hardware development.

A definition of the term 'component' in the context of hardware development can be given analogously to the definition in the context of software development, just replace 'software' by 'hardware'. This definition can be found in the beginning of this subsection.

Today component based development in software development does not have the status it has in hardware development. In the area of hardware development it would be unimaginable for an engineer to manufacture a component that already exists, without a very good reason. But in software development it is quite common to re-create existing software components.

There are some reasons for the fact that component based development is not used more often in software. These are sketched in the following paragraphs.

Firstly, techniques to specify the exact properties of software components are not wide-spread. Often the definition of a software component is just a description of the input and the output of that component, while a developer might also be interested in other properties, such as memory usage, reliability, speed and side-effects.

Secondly, some software developers dislike having to reuse software that they have not developed themselves, feeling that what they can make themselves is better or better fitting with the system under development.

Thirdly, there are too many interface standards for components. In the area of hardware development the number of interface standards is much smaller. While hardware interfaces mostly adhere to established standards, software interfaces, or parts thereof, are often made up by developers on the spot. According to a professor at my university the number of hardware interface standards is probably less than hundred, while the number of software interfaces is innumerable.

Fourthly, Software components are cheaper to produce than hardware components. If the production of a new software component would be much more expensive, like hardware components are, the people who have to pay for it would want to make sure that it is reusable. In the present situation most people who pay to have software developed do not feel that way about software.

Recommendations for better component based development

I have two recommendations for better component based development.

Firstly, there should be a limited set of interface standards for components, that is so complete that developers have no need to make their own interface standards.

Secondly, there should be standards for component descriptions. These descriptions should be more than just descriptions of functionality. In [Taula03] is a list of recommendation for component description fields. The writers have made groups of description fields. Different stakeholders can be interested in different groups. The groups are:

- Basic information. This group contains fields that give a basic description of the functionality and the interfaces that are used. There is also room for information about various quality attributes, among which are: Performance information, error handling and capacity.

- Technical details. This group contains fields that define, among other attributes, details about the execution environment of the component (operating system, virtual machine, etc.), format (source code, binary, etc.) and restrictions on usability.
- Acceptance information. This group contains fields that give information on the testing of the component. This information can be test criteria for the component, a description of test methods, test results, a test environment or test cases.
- Support. This group contains among other attributes installation instructions and contact information.

Thirdly, users of components should inform themselves better on the subject of component based development. Then they can make sure they use components correctly and that they have realistic expectations of component based development.

7.2 Advanced web services

Dr. Marc Lankhorst predicts that web services will gain a more important role in software development in the near future. A **web service** is a component that runs on a server and that can be accessed through internet protocols [wikiWebS].

Because a web service is a kind of component, the problems and challenges of components are the same for web services.

Web services can make software development more efficient in the future for the same reasons as mature component based development, because a web service is a kind of component. See section 7.1 on components.

Lankhorst predicts that in the future, developers will strive to make applications by combining behaviour from web services, instead of making a new application from scratch.

Furthermore, says Lankhorst, there will be tools that allow developers to completely describe an application using models in the form of diagrams. These tools will be able to generate applications from these models by combining available web services. Diagrams are not the ideal form of expression for describing behaviour, but that will not be a problem because most of the behaviour description is already coded in the web services.

Making applications by combining web services already happens to some degree, especially in large organisations like banks and insurance companies. These kinds of organisations often consist of multiple sub-organisations (often as a result of mergers) that share a common back office. This back office can run a set of web services that are accessed by the sub-organisations.

In the future developers will more often use web services from other parties, according to Lankhorst. This imposes some organisational challenges. Suppliers of web services will have to determine how they will charge for use. Descriptions of the properties of a web service will have to be in a standardised format, because any developer will have to be able to use it. For these properties a format can be used like the one I have described in section 7.1, because a web service can be considered to be a kind component. Only, the format described section 7.1 would have to be extended with properties specific to web services.

7.3 Model Driven Architecture

The Model Driven Architecture (MDA) is a terminological framework of which the creators believe that it will revolutionise the way software development can be done in the future. Part of the promise of MDA is that MDA will enable a new kind of software development where models serve as the eventual specification, instead of program code. If this promise can be fulfilled, it would possibly make software development more efficient.

MDA is discussed in two other places in this thesis. Section 6.3 briefly discusses Rob Vens' wish for MDA in combination with a business centred architecture. And MDA, and the promise of MDA are discussed elaborately in chapter 8.

7.4 Method engineering

Method engineering is the discipline of creating a software development method that is custom made for a specific situation. Several researchers are looking into ways of partly or completely automating method engineering.

Method engineering can make software development more efficient because it makes the activity management take less time, and the process can potentially benefit if the development process fits the development situation better.

Method engineering is discussed in two other places in this thesis. Section 6.2 briefly discusses method engineering as a wish of Ben Flokstra. Section 9 elaborately discusses method engineering.

7.5 A yet unpredictable paradigm shift

Maarten Steen did the following prediction for the future of software development: In maybe twenty or thirty years from now software development will be fundamentally changed, but in a way that we cannot predict right now.

According to Steen, today developers still used third generation programming languages. Indeed, the third generation languages Java, C++ and C# are popular languages today. In the future the need for software will be so great, that programmers will not be able to keep up any more. As a reaction to this problem, a paradigm shift will occur, a jump to a new level of abstraction.

Steen does not know if models like we know them today will be used for automatically generating code, or if maybe a declarative programming language will be used, or if someone will come up with an entirely different way of describing a software system.

Admittedly the prediction 'Something might happen that I cannot predict' may not sound like a useful prediction at first glance. But it is important to realise that the future can be unpredictable, and that software development in thirty years will be different from what we can imagine today.

7.6 Conclusion

In this chapter I have tried to answer the sixth research sub question, 'What new developments can possibly lead to a more efficient development process in the future?'

More mature component based development is a possible development that depends on standardisation of interfaces, standardisation of documentation, and awareness of what component based development is. There are no technical problems that hinder this development.

Like mature component based development, advanced web services is a development that does not depend heavily on technical developments, but mostly on organisational ones.

MDA is a terminological framework of which the creators believe that it will become very important. I have only mentioned MDA in this chapter. An elaborate discussion of MDA can be found in chapter 8.

Method engineering is the discipline of creating a software development method that is custom made for a specific situation. Chapter 9 contains a dis-

cussion of this development, that could become important in the future.

The fact should be considered that in software development, like in any field of work, developments occur that can not be predicted. We will probably not be surprised by developments that occur a year from now, but we can not accurately predict how software will be developed in twenty years.

More mature component based development and more advanced web services can make software development because the speed of development and quality of software can be improved. Method engineering can make software development more efficient because it makes the activity management take less time, and the process can potentially benefit if the development process fits the development situation better.

8 Model Driven Architecture

This chapter helps answer two research sub questions: 'What new developments can possibly lead to a more efficient development process in the future?', and 'What wishes for improvement in software development are expressed by software developers?'

The Model Driven Architecture (MDA) is a development that at first glance seems to be a development that can lead to a more efficient development process in the future. Besides that, Rob Vens mentioned MDA, in combination with his vision of a business centred architecture, as a wish for improvement of software engineering (see section 6.3).

MDA seems to be important for the future of software development. Colin Atkinson says in [Atkinson]: *'Model Driven Development as embodied by the MDA, and ontology-based knowledge representation as embodied by the semantic web, are the IT industry's primary visions for the future of software engineering and the World Wide Web respectively.'* The creators of MDA, the OMG, think MDA will have serious consequences on the way software is developed, as is demonstrated in section 8.2. Also, as explained in section 6.3, Rob Vens considers MDA to be important for the future of software development.

This chapter explains about what Model Driven Architecture by OMG is, and briefly discusses the practical implications of this theory.

This section uses the MDA Guide [OMG03] as primary source.

8.1 MDA in a nutshell

To give the reader an idea of what MDA is I will give a brief overview of MDA, without giving any definitions.

The process of making models with other models as a basis can be called 'transforming models'. The transforming of a model can happen in an informal way, by a software developer who makes a model, and just makes another one while having the first one in mind. The transforming of a model can also happen in a very formal way, by using models defined in formal languages, and predefined transformation procedures. In MDA a model transformation is the transformation of a model that is independent of a certain platform, to a variant of that model that is dependent on that platform. A transformation can be, for example, the transformation from an algorithm to the implementation of that

algorithm in Java.

MDA is a theoretical framework that facilitates reasoning about models and transformations.

8.2 Promise of MDA

The MDA Guide describes what the promise of MDA is. In this section I quote this promise in full, and explain the implications of this promise.

'This is the promise of Model Driven Architecture: to allow definition of machine-readable application and data models which allow long-term flexibility of:

- *implementation: new implementation infrastructure (the 'hot new technology' effect) can be integrated or targeted by existing designs*
- *integration: since not only the implementation but the design exists at time of integration, we can automate the production of data integration bridges and the connection to new integration infrastructures*
- *maintenance: the availability of the design in a machine-readable form gives developers direct access to the specification of the system, making maintenance much simpler*
- *testing and simulation: since the developed models can be used to generate code, they can equally be validated against requirements, tested against various infrastructures and can used to directly simulate the behaviour of the system being designed.'*

Apparently MDA will enable a new kind of software development where models serve as the eventual specification, instead of program code

From each part of the promise a statement about MDA can be derived:

- the introduction: MDA enables the creation of machine-readable application and data models.
- implementation: MDA enables the creation of models that are defined independent of implementation infrastructure.
- integration: This part of the promise is vague. It seems that MDA enables the automatic converting of data types.
- maintenance: MDA enables defining a system completely in models, as opposed to in program code (note that this implies that program code is not design).

- testing and simulation: MDA enables the creation of executable models.

This promise is reviewed in section 8.7.

8.3 Concepts that make up MDA

The MDA Guide does not give a definition of what MDA is. Rather, it gives definitions for a collection of concepts that are apparently part of MDA.

The concepts that are central in MDA are platform independent model (PIM) and platform specific model (PSM), and to a lesser degree the computation independent model (CIM). The network of definitions from the MDA Guide that lead up to the definitions of the PIM and PSM are a bit complicated and are easier understood with a little introduction: PIMs and PSMs are views from different viewpoints, with different degrees of dependence on a platform. I explain all underlined concepts, and some related concepts, in the subsections of this section.

Models in MDA

The definition of the term model in the MDA Guide [OMG03] is not essentially different from my definition from chapter 2, it is only a bit less specific:

'A model of a system is a description or specification of that system and its environment for some certain purpose. A model is often presented as a combination of drawings and text. The text may be in a modelling language or in a natural language.'

Platforms

To give the reader a feeling for what a platform is I will first give a list of examples of platforms that the MDA Guide gives, before discussing definitions.

According to the MDA Guide a platform can be generic, technology specific or vendor specific. Unfortunately these terms are not defined in the MDA Guide. Only examples of these types are given.

Examples of generic platforms that the MDA Guide gives are:

- object, which is the platform of object orientation. To avoid confusion I will call this platform 'object orientation';
- batch, *'a platform that supports a series of independent programs that each run to completion before the next starts'*.

Examples of technology specific platforms that the MDA Guide gives are:

- Java 2 Enterprise Edition [J2EE];
- Corba [CORBA].

J2EE and CORBA are only platform specifications and not implementations.

Examples of vendor specific platforms that the MDA Guide gives are:

- CORBA implementations such as Borland VisiBroker and Iona Orbix;
- J2EE implementations such as IBM WebSphere and BEA WebLogic.

The MDA Guide gives the following definition for the term 'platform':

'A platform is a set of subsystems and technologies that provide a coherent set of functionality through interfaces and specified usage patterns, which any application supported by that platform can use without concern for the details of how the functionality provided by the platform is implemented.'

There are two problems with this definition. Firstly, the definition excludes generic platforms without an implementation, even though the MDA Guide gives object orientation, which is a generic platform. Secondly, the part 'a platform is a set of subsystems and technologies' implies that a platform is defined by its implementation, and not by the 'interfaces and specified usage patterns'. This would mean that two different implementations of Java (same version of Java) are not the same platform. This is incorrect, a platform should be defined by its interfaces and specified usage patterns, and these *may* be implemented by subsystems and technologies.

I propose the following definition for the term platform, which is consistent with the examples the MDA Guide gives as platforms:

A **platform** is a coherent set of interfaces and specified usage patterns, which may or may not be implemented by a set of subsystems and technologies.

The term metamodel is often mentioned in literature about modelling and MDA. Here is a definition:

A **metamodel** is a model that describes a modelling language. For example, the metamodel for UML is a model that describes entities like 'Class', 'Object' and 'dependency'. This means that a metamodel is a kind of description of a platform, because a modelling language is a platform.

It is confusing that the writers of the MDA Guide seem to be in two minds about what a platform is. In the chapter 'Using the MDA Pattern', a few chap-

ters after the definition of the term platform, the MDA Guide says:

'What counts as a platform is relative to the purpose of the modeller. For many MDA users, middleware is a platform, for a middleware developer an operating system is the platform. Thus a platform-independent model of middleware might appear to be a highly platform-specific model from the point of view of an application developer.'

So, the MDA Guide says that something can be a platform in one context, and not be a platform in another.

Platform dependence

The MDA Guide defines platform independence, but not platform dependence. Nevertheless the terms 'platform dependent' and 'platform dependence' are used in the MDA Guide, for example in the sentence 'The PIM is platform independent because it is not dependent on any particular platform of that class.'

Because the PIM and PSM are important concepts in MDA, and platform dependence is not explained in the MDA Guide, I have chosen to explain in this subsection what I think the term means.

An important aspect of platforms is that models can be defined in terms of them. A model is always defined in terms of a modelling language, which is a platform. Using that modelling language a model may refer to other platforms.

An example of a model that depends on two platforms is a model written in UML (the Universal Modelling language, see [UML]) that models a C program. That model depends on the UML platform and the C platform. Because the references to elements of the C language are expressed in UML, some kind of mapping has to exist from UML to C. This mapping can be formal, informal, or possibly even only mental.

I propose the following definition for the term 'platform dependence:

A model is **dependent** on a platform in either of the following two cases: The platform is a modelling language and the model is expressed using that modelling language, or the model refers to elements of the platform.

Platform independence

Platform independence is a tricky term. An obvious and seemingly correct definition of a platform independent model is: 'A model that is not dependent on any platform.' But, because every model depends at least on the platform that is

the modelling language it is made in, complete platform independence does not exist. But what does platform independence mean then, if it is not the depending on a platform? The MDA Guide defines the terms 'platform independence' as follows:

'Platform independence is a quality which a model may exhibit. This is the quality that the model is independent of the features of a platform of any particular type. Like most qualities, platform independence is a matter of degree.'

I find the definition of platform independence vague, because of the part 'independent of the features of a platform of any particular type', because every model is dependent of at least one platform. It is clear in the definition is that platform independence is a matter of degree. So, one could say about two different models A and B that model A is has a greater degree of platform independence than model B. The MDA Guide does not explain how the degree of platform independence of a model can be determined, or even how it can be determined which of two models is more platform independent than the other.

There appears to be a common understanding of the term platform independence. Because intuitively, a procedure for calculating π using pseudo code feels more platform independent than a procedure for calculating π using C code, even though they both depend on one platform. The first procedure depends on the platform of natural language, math, and some conventions for writing pseudo code. The second depends on the platform of C code. Both models (remember that we regard any description of a system as a model, including code), if correct, contain a complete description of a procedure to calculate π. Why do we consider the procedure in pseudo code as more platform independent than the procedure in C? It is because, probably, the C procedure contains logic for 'technical' issues like array manipulation and memory management, while the pseudo code procedure only contains logic that we consider as important to the problem.

One might say that true platform independence exists, but only in the heads of developers. An idea about a system in the head of a developer is dependent on no platform (except maybe the developer's head?), but as soon as he expresses this idea in some language, the idea has become a description of a system that is dependent on the platform that is the language.

The term platform independence can be used in two different contexts. Firstly, the term platform independence can apply to a model. Secondly, the term plat-

form independence can apply to a software system. I can give an absolute definition for the term platform independence in the first context, in the second context I can only give a definition containing relative terms, and thus not an absolute definition.

Platform independence is a property that a software system can have, that indicates that it can be used with multiple platforms of similar type. Platforms of similar type are for example operating systems, network types or programming languages. In this context a software system is an application, a component or a software standard. So, a program written in Java can be considered platform independent because it can be used with multiple operating systems.

Platform independence is a quality that a model can exhibit to some degree. This is the quality that the model has a degree of independence from any platform from a given class of platforms. For example, a model that is totally platform independent with respect to operating systems contains no references to any type of operating system.

Views and viewpoints

The MDA Guide defines views and viewpoints as follows:

'A viewpoint model or view of a system is a representation of that system from the perspective of a chosen viewpoint.'

'A viewpoint on a system is a technique for abstraction using a selected set of architectural concepts and structuring rules, in order to focus on particular concerns within that system. Here 'abstraction' is used to mean the process of suppressing selected detail to establish a simplified model.'

The definition of the term 'view' is bad for the following two reasons. Firstly, it deviates from the established nomenclature. 'Representation' appears to mean the same as 'model', and therefore the word 'model' should be used. Secondly, the 'from the perspective of' part can be formulated clearer. I propose the following definition:

A **view** of a system is a model of that system containing only the information about that system that is relevant from the perspective of a chosen viewpoint. There can be multiple views from the same viewpoint.

The definition in the MDA Guide of the term 'viewpoint' is bad, for the following two reasons. Firstly, it defines a viewpoint as a 'technique', which it clearly

is not. And secondly, because it depends on the term 'architectural concepts', which is not defined in the same article and which is so generic that it can mean anything.

Erik Proper gives in [Proper02] a definition for the term 'viewpoint', which is too elaborate for the purposes of this thesis. I propose an shortened and augmented version of Erik Proper's definition:

A **viewpoint** is a specification of the conventions for constructing and using views. These conventions can be anything, and include at least conventions about what to model and modelling. Conventions can also include ways to present and use the views.

Central in MDA are these three basic kinds of view:

- A computation specific model is a model from the platform specific is a view of a system from the computation specific viewpoint.
- A platform independent model is a view of a system from the platform independent viewpoint.
- A platform specific model is a model from the platform specific is a view of a system from the platform specific viewpoint.

These three views are explained and defined in the following sections.

Computation independent model

The computation independent model (CIM) is defined as follows in the MDA Guide:

*'A **computation independent model** is a model of a system that shows the system in the environment in which it will operate, and thus it helps in presenting exactly what the system is expected to do.'*

The CIM appears to be the MDA term for functional model or context model.

Platform independent viewpoint

The MDA Guide gives the following definition for the term 'platform independent viewpoint':

'The platform independent viewpoint focuses on the operation of a system while hiding the details necessary for a particular platform. A platform independent view shows that part of the complete specification that does not change from one platform to another.'

This is a bad definition, for the following reasons:

Firstly, it defines a viewpoint as if it were a model, which it is not (see section 0 about viewpoints).

Secondly, the first sentence says very little, because almost every model focuses on the operation of a system and lacks information that may be part of a more detailed model. Probably the writers of the MDA Guide want to say that the platform independent model contains mostly information that is regarded as essential by the person who made the model, and little information that is added to make this essential idea work on a chosen platform.

Thirdly, the second sentence of the definition makes an implicit reference to the process of transforming a model to another model that works on a different platform (or set of platforms), without any clarification. What exactly does it mean that a part of a model remains the same when that model is transformed to another platform? What platforms are we talking about anyway? These questions are not answered in the MDA Guide.

I propose the following definition for the term 'platform independent viewpoint':

The **platform independent viewpoint** is a viewpoint that focuses on aspects of a system that are independent of a given class of platforms.

Platform independent model

I accept the definition of the term platform independent model from the MDA Guide, which is the following:

*'A **platform independent model** (PIM) is a view of a system from the platform independent viewpoint. A PIM exhibits a specified degree of platform independence so as to be suitable for use with a number of different platforms of similar type.'*

The 'different platforms of similar type' need some extra explanation. I will not try to give a definition of the term platform type. Examples of platform types are operating systems and component communication platforms.

Platform specific model

This section explains what the platform specific model (PSM) is. In order to do that the term platform independent viewpoint is explained first.

Platform specific viewpoint

The MDA Guide defines the term 'platform independent viewpoint' as follows:

'The platform specific viewpoint combines the platform independent viewpoint with an additional focus on the detail of the use of a specific platform by a system.'

This definition seems more like a definition for a kind of model than a definition for a kind of viewpoint. I propose the following definition:

The **platform specific viewpoint** is a viewpoint that focuses on aspects of a system that are dependent of a given class of platforms.

Platform specific model

Having said that the every platform depends on at least one platform, the term 'platform specific model' (PIM) might seem a bit useless.

The MDA Guide gives the following definition, which I accept:

*'A **platform specific model** [PIM] is a view of a system from the platform specific viewpoint. A PSM combines the specifications in the PIM with the details that specify how that system uses a particular type of platform.'*

This definition means that a PSM relative to a PIM, and that a PSM depends on a platform that its PIM did not depend on.

Platform model

A platform model is a description of a platform. The MDA Guide gives the following definition, which I accept:

*'A **platform model** provides a set of technical concepts, representing the different kinds of parts that make up a platform and the services provided by that platform. It also provides, for use in a platform specific model, concepts representing the different kinds of elements to be used in specifying the use of the platform by an application.'*

If a platform model is expressed in a formal language, it can be used by a software development tool to assist in making models that depend on that platform. For example, UML [UML] supports the concept of profiles. A UML profile can define the aspects of a platform that can be modelled in UML. A simple application of profiles is letting a software development tool check if a UML model that depends on a certain platform adheres to the platform model of that platform in a UML profile.

A platform model is a model, so it must depend on other platforms. The descriptions of these other platforms are platform models depend on other platforms as well. These dependencies can not go on forever. If one follows this tree of platforms that are used to define other platforms to its root, one would find the platform of natural language, which can only be defined in other natural language. This is not a problem, but developers should be aware that it is very difficult if not impossible to nail down the definition of a platform absolutely. The best situation a developer can hope for is that the model he makes is understood by its audience (possibly a computer) exactly the way he understands it himself. And I will leave the question of what 'understanding' means exactly to philosophers.

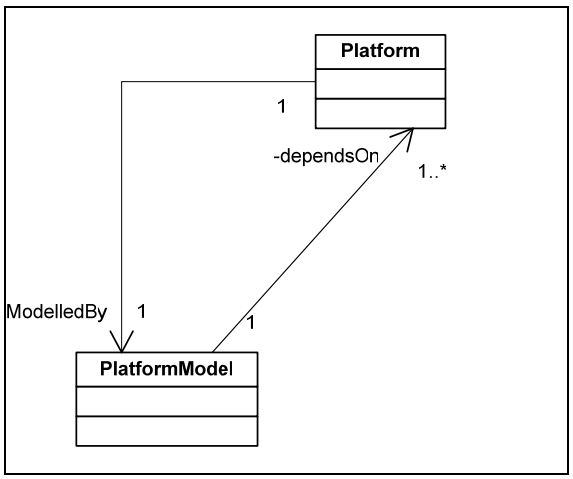


figure 5: A platform is modelled by a platform model, which depends on platforms, which are modelled by platform models, which depend on… etcetera.

Transformations

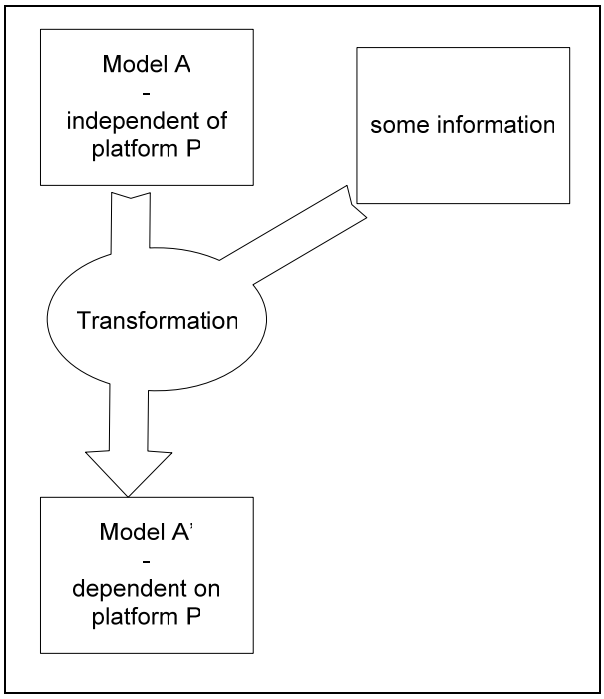


figure 6: The MDA pattern – A platform independent model is transformed into a platform specific model, using some extra information that is provided to the transformation.

In general a transformation in MDA is the following: A model that is independent of a given platform P (the PIM) is transformed, using some extra information, to a new model that is dependent on platform P (the PSM).

A **mapping** is a description of a procedure to transform a PIM into a PSM.

The MDA Guide names the following approaches to model transformation:

A first approach is to define a mapping from the metamodel of the PIM to the metamodel of the PSM. In principle this is the same process as the translation of any other computer language into another.

A second approach is to prepare the PIM before it is transformed, by applying marks. A mark is an addition to a PIM, that can be used in a mapping. For example, one might add a mark to a PIM to indicate that objects of a class called Customer should be persistent. Although marks provide additional description of the system being modelled, they should not be considered as part of the PIM, because they can be specific to the PSM that is being created with a mapping. These marks can be used in combination with any other approach to model transformation.

A third approach is to make a PIM that contains classes that are subclasses of more generic superclasses. A mapping is made to transform these superclasses to versions of those classes that are fit for some platform.

A special kind of transformation is needed when two models have to be merged to make a model that combines the information from the two models. Merging is needed in any situation where there are multiple models that describe different aspects of the same software system. The merging process is trivial if the multiple models describe different parts of the software system. It is not trivial if the multiple models describe different aspects of a single part of the software system. A obvious example is the situation where one model describes the user interface of a software system, and other models describe the rest of the system. Unfortunately the MDA Guide gives no information on how to achieve the merging of models.

A mix of the abovementioned approaches can be used to achieve a transformation.

Transformations can be performed manually, automatically, or half-automatically. A manual transformation can be performed without consciously making a transformation; making program code while looking at a model can be considered a manual transformation. An example of a half-automatic transformation is a transformation where a PIM is extended with marks, and where a tool uses these marks to make a transformation. An automatic transformation is a transformation where a model is transformed by a tool that needs no additional information. An example of this is the compiling of a computer program, where program code is transformed into machine code.

Ideally, a record is made of all transformations in a software development process. This record can consist of all models that are made during software development, and descriptions of how transformations were achieved.

8.4 The problem with the MDA Guide

A problem with MDA is that its defining document, the MDA Guide, is badly written.

This fact is illustrated by my comments in section 8.3 of this thesis about various definitions from the MDA Guide that are vague or contradicting with other definitions. Also, both sections 3.10 and chapter 4 of the MDA Guide are on transformations, and their content overlaps. And chapter 6 is called ‘Using the

MDA pattern' and is devoted mostly to redefining the term 'platform'.

Apart from that, there are quite some smaller mistakes. Some examples are: on page 6-5 a quotation is made without mentioning the source; page 2-6 contains a reference to a wrong paragraph.

A likely cause of this messiness MDA Guide is the fact that 45 people from 32 different companies have worked on this document, which consists of only 62 pages.

8.5 Modelling style and MDA

MDA encourages thinking of a software development process as a sequence of creations and transformations of models. This sequence goes on until there is a model that can be executed or transformed to a model that can be executed. In this way of thinking the distinction between modelling and programming has become less relevant. Does this mean that programming has become as easy as modelling, or has modelling become as difficult as programming?

Modelling in a MDA-inspired software process is different than in a classic software development process. MDA encourages the creation of formal models. A formal model is a model that is defined in a formal language. The other kind of model is the informal model, which does not have adhere strictly to a modelling language, as long as it can be understood by the creator and target audience.

Some reasons why formal models are more useful than informal models are the following: formal models can potentially be transformed automatically or half-automatically; formal models can potentially be automatically checked; formal models that describe a complete system can potentially be executed; formal models can potentially be integrated with other formal models that describe the same system.

A reason why informal modelling is more useful than formal modelling is that informal models generally allow the creator a greater degree of expression, because he does not have to adhere strictly to the restrictions of a modelling language. Because of this, informal modelling can be done faster, and it is easier to express ideas that can only be expressed in a formal modelling language with great difficulty or not at all.

Generally, formal modelling should be done with models that are intended for use in automatic or half-automatic transformations. Informal models should be

used in all other cases. Especially when a system is modelled on a conceptual level, strictly obeying the laws of a modelling language is wasted effort.

In the beginning of this section I stated that MDA encourages thinking of a software development process as a sequence of creations and transformations of models. Because of that, it can also encourage developers to think more on the question what information about the system under development should be placed in what model. Some questions to consider when making this decision are the following:

- What models should be dependent on which platform?
- How much level of detail is needed at the different stages in the development process?
- What mappings are available?
- Who needs what information?
- Which modelling languages allow the developer to describe aspects of the system most conveniently or efficiently?

With respect to the last question I should note that a programming language can be considered a modelling language.

If the system under development is a member of a class of similar applications it is possible to make simple models with little information about behaviour, that are still complete descriptions of the system. Consider for example an application that consists of an input screen, presented to the user via the internet in a browser, where the user can enter information, that the application saves in a relational database. This application can be generated with a major database application, that takes a simple application description (which is a model) as input.

Some modelling languages allow complete descriptions of systems on higher levels of abstraction than others. For example, Java lets the developer just specify an automatically resising list with one command, while the developer needs to write more to achieve the same in C. Similarly, some modern modelling languages allow developers to ignore considerations like distribution of the software over multiple machines, and choice of data-structure. The good news for programmers on lower levels of abstraction is, that eventually all software runs on processors, and that programming needs to be done on every level of abstraction.

8.6 Perceptions of MDA

Nowadays MDA is very popular. A Google search for 'Model Driven Architecture' yields about 153.000 results. Various institutes offer courses and seminars that are completely on MDA (for example [MDACourse1], [MDACourse2], [MDACourse3]). Various tool vendors advertise using MDA in some way in their tools (for example [Arcstyler], [SelectMDA], [AndroMDA]).

Various articles make assumptions about MDA that can not be verified. Some *examples:*

'The MDA supports many of the commonly used steps in model driven component based development and deployment. A key aspect of MDA is that it addresses the complete life cycle analysis and design, programming aspects (testing, component build or component assembly) as well as deployment and management aspects.' and 'Historically, the integration between the development tools and the deployment into the middleware framework, has been weak. This is now beginning to change by using key elements of the MDA -- specific models and XML DTDs that span the life cycle, and profiles that provide mappings between the models used in various life cycle phases.' [xmlCoverMDA]

'MDA is a new software architecture principle in the sense that it follows, incorporates or supports most industry standards. It enhances the capability of the software architecture to be flexible, extensible and adaptable to new technologies or standards, and to abstract business application components in the early stages of EAI initiatives. Most importantly, MDA enables business experts and software engineers to focus on business risks, issues and concerns, ensuring the effective use of existing and evolving technologies.' [Hazra]

Rob Vens said that MDA is important for his vision of a business centred architecture (see 6.3). But the only principle from MDA Guide that is important to his vision is that of PIMs and PSMs. In his vision a complete organisation is captured in models, which are so complete that these models can be used for software right away, without making a transition. Applications made in this way are PSMs, specific to some technical platforms, the models of the organisations are PIMs, that are only dependent on a modelling language. Vens had this vision already before the OMG has made MDA public

Many people use MDA for different meanings. This is possible because, as I have demonstrated in section 8.4, the defining document of the MDA, the MDA Guide is badly accessible. Someone who has taken the effort of working his way

through this document will not have a clear idea of what MDA is, but rather a collection of impressions.

For a reason I do not know MDA, however vaguely defined, has an appeal to people, even if they do not know exactly what it is. Also, because MDA has become buzzword, some people probably reference MDA because buzzwords attract attention to their ideas.

8.7 The promise of MDA reviewed

This section reviews the 'promise of the MDA' as listed in section 8.3, part by part.

'to allow definition of machine-readable application and data models' – The MDA Guide does not explain the creation of machine-readable application and data models. Besides, every application and data model that can be created on a computer is already machine readable.

'implementation: new implementation infrastructure (the 'hot new technology' effect) can be integrated or targeted by existing designs' – In section 8.2 I have said that this part of the promise implies that MDA enables the creation of models that are defined independent of implementation infrastructure. This is not the case. A new implementation infrastructure can be regarded as a platform. Say that a system already has a PIM called M, and a PSM called M'. If a new platform is of the same platform class as M' (for example, they are both programming languages), the system can be made to use the new platform by making a mapping from the PIM to the new platform.

'integration: since not only the implementation but the design exists at time of integration, we can automate the production of data integration bridges and the connection to new integration infrastructures' – In section 8.2 I have said that this seems to imply that MDA enables the automatic converting of data types. Unfortunately the terms 'data integration bridge' and 'integration infrastructure' are not defined in the MDA Guide, and these are not generally known terms. In any case, I have seen no indication that MDA enables automatic conversion of data types.

'maintenance: the availability of the design in a machine-readable form gives developers direct access to the specification of the system, making maintenance much simpler' – In section 8.2 I have said that this part of the promise implies that MDA enables defining a system completely in models, as opposed to in program code. The implicit claim is also that MDA enables the creation of models that are useful as design models and are also executable. Currently work is being done in the area of executable models that are made with an extended version of UML (for example iUML, see

[iUML]). But my impression is that the biggest win of these kind of tools is that the activity that is originally called programming is more integrated with modelling, but has not disappeared. Calling the creation of these kind of tools a fulfilment of the promise of MDA is perhaps claiming someone else's victory. Though it is true that the theory of MDA can help reasoning about these kind of tools.

'testing and simulation: since the developed models can be used to generate code, they can equally be validated against requirements, tested against various infrastructures and can used to directly simulate the behaviour of the system being designed.'– In section 8.2 I have said that this part of the promise implies that MDA enables the creation of executable models. Like the previous promise, this is not really a promise of MDA, but of tools that support executable models.

In conclusion, the promise that the MDA Guide does about MDA is a hollow promise. None of the points of the promise can be fulfilled by MDA on itself, as it is defined in the MDA Guide.

8.8 Conclusion

This chapter helps answer two research sub questions: 'What new developments can possibly lead to a more efficient development process in the future?', and 'What wishes for improvement in software development are expressed by software developers?'

MDA is described in a document called the MDA Guide. This document is badly written, and MDA is not really defined in it. The guide is a collection of definitions of concepts, co-written by many different writers.

The central concepts from MDA are those of platform independent model (PIM) and platform specific model (PSM). A PIM is independent from a certain plat-

form, and PSM is dependent on a certain platform. A PSM can be generated by applying a model transformation to a PIM. The concept of separation of PIM and PSM resembles the concept of separation of design and implementation of a system, which is not new.

Even though MDA is badly defined, it has become a buzzword. Many people seem to have many different ideas about what it actually is.

The promise of change in software development that is made in the MDA Guide is spectacular. It sketches a practice of software development where various parts of the software development are automated. Unfortunately MDA, as defined in the MDA Guide, can not make this promise real.

9 Method engineering

This chapter helps answer two research sub questions: 'What new developments can possibly lead to a more efficient development process in the future?', and 'What wishes for improvement in software development are expressed by software developers?'

Method engineering is mentioned by Ben Flokstra as a wish for improvement of software engineering. Besides that, it is a development that can lead to a more efficient software development process in the future. Method engineering can make software development more efficient because it makes the activity management take less time, and the process can potentially benefit if the development process fits the development situation better.

Method engineering is the discipline of creating a software development method that is custom made for a specific situation. This chapter will show what method development is, how it can be used, and how it can enable more advanced tool support for the software development process.

The first part of this chapter is an introduction to method engineering, the second part of this chapter is a review of a book on method engineering [HarmsBrink02], the third part of this chapter is a review of an article on method engineering [KarlsÅger04], the fourth part of this chapter is a discussion on Ben Flokstra's thoughts on method engineering, and in the conclusion of this chapter are my thoughts on the status of method engineering today.

9.1 What is method engineering

Every software development process is performed differently. Often a standard method is used, that is adapted to suit the needs of the stakeholders in the development process. Method engineering is the discipline that takes a scientific approach to the tweaking and adapting of standard methods. There is no single method for method engineering, just as there is no single method for software development.

I will use the definition from [KarlsÅger04], that I have rewritten to make it independent from its context:

*'A **method** consists of three parts:*

- *a process to guide the performance of activities*

- *a notation for documenting the results of the activities*
- *a set of concepts used to describe the problem domain'*

In the discipline of method engineering a software development method is regarded as a collection of separate fragments that are combined to make a method. Such a fragment can be anything the method engineer finds useful. Some random examples are: 'make a domain model', 'domain model', 'developer', 'deliverable', 'Rational Rose'. A method that is created using method engineering is tailored to a specific project situation, and is therefore often called a situational method.

Here follows the general approach to a software development project using method engineering is the following. It is an abstraction I made from [HarmsBrink02] and [KarlsÅger04].

- characterise the project
- select method fragments from a database of method elements
- assemble the method elements to create a situational method
- execute the project using the situational method
- use experiences from the project to enhance the database of method fragments

9.2 Method engineering and internet

Sjaak Brinkkemper and Frank Harmsen have written a book called Method Engineering [HarmsBrink02]. This section summarises and reviews this book.

The book Method Engineering describes a small framework of terminology for use with method engineering. Its purpose is for the reader to get a basic understanding of what method engineering is, not to enable the reader to practice method engineering himself right away. Then some descriptions of uses of method engineering in practice follow. The book ends with an evaluation. I think the most important message of the book is that a method can benefit greatly from being managed and used via the internet.

Terminology and the process

The process of method engineering that is described by the book is a more specialised version of the process described in section 9.1, that is not directly usable in practice. A new element that is introduced is the method bank.

A method bank is a database in which all method fragments are stored. How this is done is left open. Examples of additional kinds of information that can be saved in a method bank are assembly rules, semantic information on the method fragments, assembly rules and text on previous experiences and best practices.

Assembly rules govern how method fragments should be assembled. They can be formal or informal. An example of an informal assembly rule is 'if many users believe there is a risk for conflicting demands, a facilitated workshop is needed'. Ideally, a set of method fragments and assembly rules guarantees that the situational method is complete, consistent, efficient, correct and applicable.

The process for method engineering from the book Method Engineering is described as follows:

- The project is characterised.
- Using the method bank and the characterisation of the project, a selection of method-fragments is made.
- In the assembly stage the method fragments are assembled, which results in a situational method that may or may not be complete.
- The situational method is applied.
- Experiences from the people involved are collected, analysed and used to improve the method bank.

Potential for tool support

The process for method engineering described above does not strictly need any tool support, but it is probably not feasible to perform it properly by hand. The book describes a list of actions that are best supported by tools. Here is the list:

- Project characterisation. This can for example be done through an automated questionnaire.
- Recording method fragments.
- Making method fragments accessible. This is probably best done through the internet.
- Selection of method fragments. This can be partially automated using the results from the project characterisation.
- Manipulation of method fragments.

- Assembling the method fragments.

There already is a program called Scenario Analyser that supports characterising projects and the selection and assembly of method fragments. It has knowledge of several DSDM development scenarios. These are method descriptions that are adaptations of the popular method DSDM [DSDM]. Examples of development scenarios are 'standard DSDM' and 'waterfall development'. The Scenario Analyser lets the user fill in a list of multiple choice questions about the project that is about to start. Using those answers, it gives a number for each development scenario it knows, indicating its suitability. It also gives some additional advice.

The internet (or intranet, which I will consider as the same thing in this case) can be used for supporting several or all of the activities mentioned above. A internet application could be build around a method bank. This application and the method bank can be maintained in one place, and used by everyone.

The book Method Engineering describes a case from Capgemini on the use of internet to make their method accessible. Capgemini has offices in multiple countries, and wanted a method that had the right mix between standardisation and flexibility for their development method. A standard method, an application and a web interface were developed that allowed the standard method to be adapted within certain boundaries by local offices.

The book Method Engineering also contains cases from Baan and ABN Amro, on how those companies made methods accessible through the internet, and on how these methods could be adapted quickly through feedback from users.

The writers of Method Engineering speculate that in the future frameworks might be built that for the brokering of message fragments, analogous to message brokers that are already available for software components. A method fragment broker could try to find appropriate method fragments from various sources, given a project characterisation.

Conclusion

The book Method Engineering is a useful book that gives an insight in the basics of method engineering. The cases in the book are all primarily focused on making a method better accessible through the internet. These cases can be seen as method engineering, because they describe how they methods can be adapted to suit the needs of users. But the adapting of methods is not really

new, it is only made easier by the efficient use of the internet. Unfortunately, the writers did not relate the case descriptions to the framework that is sketched in the beginning of the book. For example the process of building a method using a method bank, method fragments and assembly rules can not recognised in the case descriptions.

Method Engineering is an interesting read, but has too little relation between theory and practice.

9.3 Method for Method Configuration

The Method for Method Engineering (MMC) is for method configuration, which is a kind of method engineering. It is for making a configuration of a base method. It is developed by F. Karlsson and P.J. Ågerfalk. These people developed the MMC in the course of two years in close cooperation with an unnamed large Swedish IT company.

The MMC consists of a group of related basic concepts, and a process description for using these concepts. The concepts and process are explained in the following two sections. In the section after that I explain the potential for tool support that the MMC has.

The basic concepts

*'A **base method** is the systems engineering method chosen as the starting point for the configuration work.'*

In a diagram the writers describe a base method as consisting of group of actions. In the text they say: 'Method configuration means deciding which prescribed actions of the base method should be performed and to what extent.' Apparently the writers make the simplification of considering a method as a group of actions. This may be justified, but the writers should have given this justification explicitly.

A base method is only useful in an organisation where each software development project shares a large enough set of characteristics. A base method can, for example, be a configuration of the Rational Unified Process [RUP].

*'A **characteristic** is a delimited part of a development situation, focusing on a certain problem or aspect which the method configuration aims to solve or handle.'*

A characteristic can have one or more values. An example of a characteristic is 'deployment method', and examples of the values that this characteristics can

have are 'thin client deployment' and 'local client deployment'.

*'A **development situation** is an abstraction of one or more existing or future software development projects with common characteristics.'*

A development situation consists of a collection of values of characteristics. An example of a development situation could be [thin deployment client, risk of conflicting demands]

*'A **prescribed action** is a process fragment.'*

This is a useless definition because the term 'process fragment' is not defined. It is all the more confusing that the writers call a prescribed action elsewhere a 'method fragment'. A prescribed action is just an action that can be performed in a project. A prescribed action exists in a level of granularity. Levels of granularity are described in [BriHarSae]. A prescribed action on a higher level of granularity consists of actions on a lower level or granularity, and an action from the lowest level of granularity is an atomic action.

I have rewritten the definition of the term 'configuration' to make it independent of the context in which it was written in the article.

A **configuration** is a particular variant of a systems engineering method tailored for a specific development situation—a situational method derived from one specific base method.

It is confusing that the writers now speak of 'systems engineering method' instead of just 'method', while these terms apparently mean the same. A configuration is not necessarily a complete method. A configuration is collection of prescribed actions from a base method, combined with a classification for each of those prescribed action.

A **classification** says something of the combination of the importance and the complexity of the area of concern that a particular prescribed action addresses. A classification can have the following values:

omit;		
perform reduced;	perform as is;	perform extended;
emphasise reduced;	emphasise as is;	emphasised.

'Omit' means that the prescribed action is not performed.

'Perform' means that the prescribed action has to be executed without special attention because it has normal importance.

'Emphasise' means that the action has to be performed with extra attention because it has extra importance.

'Reduced' and 'extended' are only applicable to prescribed actions that consist of actions of a lower granularity (so they do not apply to atomic actions).

'Reduced' means that the prescribed action needs to be performed, but that not all the prescribed actions of lower granularity that it is composed of need to be performed.

'Extended' means that the action is extended with extra prescribed actions of a lower granularity.

*'A **configuration package** is a configuration of the base method suitable for one specific characteristic's value.'*

A configuration package is a configuration of only that part of the base method that is relevant to the characteristic it was made for.

*'A **configuration template** is a combined method configuration, based on Configuration Packages, for a set of recurrent project characteristics.'*

From the rest of text can be derived that with 'characteristics' the writers actually mean 'values of characteristics'. A configuration template a combination of the prescribed actions and classifications from several configuration packages that is made to suit one specific development situation. If multiple configuration templates are combined that cover the same prescribed actions from the base method, the classifications of these actions can be conflicting. These conflicts need to be resolved in a configuration template.

The process

This section describes the activities that are performed to use the MMC. The activities needed to create a base method are not described, because that falls out of the scope of the MMC. I think the writers of the article should nevertheless have briefly discussed this topic. The converting of, for example, RUP [RUP] to a base method as it is defined in the article is quite a problem, because the complete method has to be converted to a collection of prescribed actions. Analogously, the article does not discuss the conversion of the output of the MMC, which is a configuration template, to a method that is usable in a working environment.

Here follow the activities that are listed as part of the MMC. They are divided in

three sections, each of which serves a single purpose. The following is a list of these sections with their activities.

First section: Creating or modifying configuration packages

The first section of activities in the MMC is for creating or modifying configuration packages. This consists of two actions:

- Abstracting past en future projects into one or more development situations
  input: Knowledge about projects.
  output: One or more development situations, each of which consists of a lists of values of characteristics.
- Creation or modifications of configurations packages.
  input: Development situations (for determining what configurations packages are needed), a base method containing a list of prescribed actions, and previous experience with the MMC.
  output: configuration packages.

Second section: Creating configuration templates

The second section of activities in the MMC is for the combining of existing configuration packages into new or modified configuration templates. This consists of two actions:

- Selecting configuration packages. These selected configuration packages can be conflicting if they contain different classifications for the same prescribed actions (classification conflicts).
  input: Configuration packages, development situations, previous experience with the MMC.
  output: Selected configuration packages.
- Combining configuration packages. Classification conflicts have to be solved, and consistency of the configuration template has to be evaluated and guaranteed.
  input: Selected configuration packages.
  output: Configuration templates.

Third section: Starting a project

The third section of activities in the MMC is for starting a specific project. A configuration template is selected, adapted to fit the project and a usable

method is created. The following actions are listed:

- Identifying project characteristics. The project is analysed and a list of values of characteristics is made.
  input: The project situation.
  output: A list of values of characteristics.
- Matching project and configuration template. Perhaps it would have been better if this action was represented as two actions, because it consists of finding a template that best matches the values of characteristics from the previous step, and then of creating a method that can be used in practice, using this template. The article takes a bit of shortcut by only describing this as follows: *'If a matching configuration template is found we fine-tune it into a situational method'*. This step depends on the assumption that the collection existing configuration templates is so well made that there is a configuration template that is almost fitting for the current project.
  input: Project characteristics.
  output: A method.

Potential for tool support

I will assess the potential for tool support in the MMC by evaluating each of the actions listed in the previous section. Some descriptions of actions are shortened or removed, full descriptions can be read in the previous section.

1. *Abstracting past en future projects into one or more development situations.*
   input: Knowledge about projects.
   output: One or more development situations, each of which consists of a lists of values of characteristics.

This can be supported by an application that facilitates the creation of new characteristics and their values and allows reuse of existing characteristics. Because the input can not be formalised the activity as a whole can not be automated.

2. *Creation or modification of configuration packages.*
   input: Development situations, a base method containing a list of prescribed actions, and previous experience with the MMC.
   output: configuration packages.

If there is some place where information is recorded on the relations between the prescribed actions, interesting applications can be built to partially auto-

mate this action.

Examples of possible prescribed action – prescribed action relations are:

- If one prescribed action is performed then another needs to be performed as well;
- Two prescribed actions are mutually exclusive;
- One prescribed action is on a higher level of granularity that another.

If this kind of information is available on the prescribed actions, a tool can be built that automatically checks the consistency of a configuration package.

If there also is information available on the semantics of the prescribed actions, an application can be built that can build a configuration package, or that can suggest prescribed actions that can be included in a configuration package. This application could use a questionnaire to ask about the characteristics of the desired configuration package, and use some kind of expert system to suggest possible configuration packages, including classifications for the prescribed actions.

Ideally, an application that automatically or semi-automatically assembles prescribed actions following the criteria a project description, also explains its own output. Then it would behave like a real expert system. With this reasoning in mind a user can more easily adapt the suggested configuration package.

3. *Selecting configuration packages.*
   input: Configuration packages, previous experience with the MMC.
   output: Selected configuration packages.

The selection of configuration packages that suit the values of characteristics of a configuration template can be automated. In order to achieve this, a database has to be built which contains pairs of configuration packages and values of characteristics. This database could be used by multiple organisations that use the MMC.

4. *Combining configuration packages. Classification conflicts have to be solved, and consistency of the configuration template has to be evaluated and guaranteed.*
   input: Selected configuration packages.
   output: Configuration templates

This action is basically the same as action number 2, because this action too consists of combining a group of prescribed actions. Only the purpose is to create a configuration template, instead of a configuration package.

5. *Identifying project characteristics. The project is analysed and a list of values of characteristics is made.*
   input: The project situation.
   output: A list of values of characteristics.

This action can be supported by an application that knows what characteristics are defined in the organisation and their values. Using a questionnaire this application can ask questions about the project, and then help make the list of characteristics and their values.

6. *Matching project and configuration template.*
   input: Project characteristics.
   output: A method

In the previous section I mentioned that this action actually consists of two actions: the matching of the values of characteristics of the project with a template, and creation of a method that can be used in practice. The matching of the values of characteristics of the project with a template can be automated. An application that automates this only has to analyse which template has most values of characteristics in common with the values of characteristics of the project.

The details of the creation of a method that is usable in practice lies outside the scope of the MMC, because it requires details about the method that is used.

Conclusion

The article of Karlsson and Ågerfalk is unclear in places, because it uses its own terminology sloppily. In some places the term 'configuration' is used where 'method configuration' is meant. The term 'prescribed action' is defined only as a 'process fragment', which, in turn, is not defined. In some places the term 'method fragment' is used, where apparently 'prescribed action' is meant. The sloppy use of terminology that gave me the most trouble was that of 'characteristic' and 'value of characteristic'. These terms are defined as separated terms, but nevertheless the writers use mostly 'characteristic' where they mean 'value of characteristic'. It also took me a while to realise that the base method is not a method like DSDM or RUP, but a representation of such a method in the form of a collection of prescribed action. These examples of sloppiness are possibly an indication of the immaturity of the theory.

It is an oversimplification of the article to model a method as only a collection of

actions. The method that is configured has to be of the kind that can be translated, without too much loss, to a collection of actions to create a base method. The other way around has to be possible too; so, is should be possible at the end of the preparation for a project to translate a collection of actions with classifications to an instance of the method. How these translations are supposed to be performed is left open in the article. However, this simplification enables the easily understandable method for method configuration that is described in this chapter.

The potential and need for tool support lies mostly in the combining of actions into chunks of method (called configuration package and configuration template in the article). How far this tool support can go is an open question, because the article says nothing about what information about actions and about the relationships between actions should be recorded.

The MMC is an interesting theory, which is currently too little specific to directly use in practice. In my description of the theory I have not essentially abstracted the theory, it is as little specific as described in this chapter. This information is interesting and inspiring, but it can not be implemented in an organisation directly.

9.4 Advanced futuristic method engineering

When I asked Ben Flokstra about his dream application that could make software development much easier in the future, he mentioned advanced method engineering.

Ben Flokstra told me that many projects already use a hybrid method that is composed of elements from different standard methods. His dream was that in the future there would be an application that lets the user enter the criteria of the project, and that automatically suggests a set of elements that together form a complete method that answers to the criteria of the stakeholders.

Note the difference between the MMC and Flokstra's dream application. The MMC is about the adapting of one method to make it fit the current project. Flokstra's dream application assembles a method from scratch.

The application that Ben Flokstra describes resembles the ideas from the section 'Potential for tool support' in section 9.3, that describes the potential for tool support that is offered by the MMC. Especially the potential for tool support that I described for the assembling of configuration packages (action number 2)

is applicable to Flokstra's dream application. Flokstra's dream application might be realised as an extension to the MMC.

This application would not take one, but several regular methods that have been converted to base methods, consisting of prescribed actions, along with relations between them as described in the abovementioned section, and semantic information.

The application would use a questionnaire to ask a user about the criteria for the development project. These criteria can be about existing methods, like: 'users need to have little experience with DSDM', or 'TESTBED diagrams have to be used to model the business domain'. The application represents these as values of characteristics, perhaps along with numbers to signify importance. The application then assembles the prescribed action from the base method to fit the criteria, using artificial intelligence. The end result is a method in the form of prescribed actions and relationships between them, and an explanation for the choices that the application made. This application could be extended to transform this method into a form that can be directly used by those involved in the project.

9.5 Conclusion

This chapter helps answer two research sub questions: 'What new developments can possibly lead to a more efficient development process in the future?', and 'What wishes for improvement in software development are expressed by software developers?'

In this chapter I have described method engineering as a new development, and I have reviewed Ben Flokstra's wish for method engineering.

Real method engineering or configuration does not seem to take place in practice yet. The book Method Engineering describes a theoretical framework for method engineering, but the example cases that are described in the book do not use this framework and describe relatively straightforward web pages that help present a method to the users.

The purpose of the Method for Method Configuration is not the engineering of a complete method, but the configuring of an existing method. This description of this method leaves a lot of questions and is not directly usable, but it is an interesting theory that might be of practical use in the future.

Flokstra's dream application is an application that can generate a complete, us-

able method that is tuned to a specific development situation, after it is given enough information about the development situation. If this application could be realised, it would speed up the project management activity, by speeding up the preparation for a software development process. Also, if the application would create a method that would be a better fit for the project situation, the management activity could be executed more efficiently, resulting in a more efficient software development process.

It is conceivable that this application could be realised by adapting the MMC. But, the MMC, as it is described in [KarlsÅger04], is far from usable.

10 Conclusion

In this chapter, the answers this thesis gives to the six research sub questions are briefly summarised. Then, I will try to give an answer to the main research question, 'How can improvement of tool support and development processes help make software development more efficient?'

10.1 The sub questions

Sub question 1: What exactly is software development, and how is it done?

In chapter 2 I have answered the first research sub question: 'What exactly is software development, and how is it done?'

I answered this question by formulating and defining a terminological framework for software development. By doing this I marked the start and the end of software development. This is important, because in practice it can be unclear when exactly a software development project has started and when it has ended. In my definition, software development starts with activities that contribute to defining the software, and ends with activities that contribute to deploying the software.

Sub question 2: Which types of inefficiency can be observed in a software development process, and how can these inefficiencies be remedied?

In chapter 3 I answered the third research sub question: 'Which types of inefficiency can be observed in a software development process, and how can these inefficiencies be remedied?'

The types of inefficiency that can be observed in a software development process are people related inefficiency, tool related inefficiency and process related inefficiency.

The interviews revealed that software developers believe the most important causes for inefficiency are people related. Inefficiency that appears to be process related is often also tool related.

Examples of tool and process related inefficiencies are:

- Unnecessary lack of insight in the relationships between artefacts. This

can be remedied by an advanced repository.

- Unnecessary effort in adapting a development method to a specific situation. This can be remedied by method engineering.
- Unnecessary complexity of software. This can be remedied by Vens' business centred architecture.
- Unnecessary repetition in software development. I have not found a remedy for this kind of inefficiency.
- Possible lack of automation by tools. Chapters 4 and 5 show that this is probably not a problem.

Sub question 3: Which actions of the software development process can be automated?

In chapter 4 I answered the third research sub question, 'Which actions of the software development process can be automated?' It appears that very few activities can be fully automated. Fortunately, automation can be used to support each of these activities.

The activities that can be automated are coding, given that all the necessary creativity has gone into the technical model, and verification, given that the requirements are formal.

Competencies that a computer can not be capable of, are competencies which require creative thinking, or the handling of non-predefined kinds.

To find out how an action can best be supported by tools, it would be necessary to dissect the action in sub actions, and analyse each of these the way I analysed the main action from the software development process in section 4.4.

Sub question 4: What degree of automation of the actions from the software development process is offered by existing tools?

In chapter 5 I answered the fourth research sub question, 'What degree of automation of the actions from the software development process is offered by existing tools?'

The results of my survey are consistent with the predictions of chapter 4, where I concluded that no activity from the software development process can be automated, except for coding (in limited circumstances), and the test activities. These activities are fully automated in the tool Statemate. However, in order to

automate these activities, the activities that produce the required input, have to be performed extra rigorously.

My findings were that comparing available tools costs far more time than I have available, mostly because of poor documentation of the available tools.

A finding that surprised me was that while the number of tools keeps increasing, the respondents of my interviews mostly used the functionalities that have been available for a long time.

Sub question 5: What wishes for improvement in software development are expressed by software developers?

In chapter 6 I have tried to answer the fifth research sub question, 'What wishes for improvement in software development are expressed by software developers?'.

I have given a list of avenues to possible improvements in tool support for the software development process. In short, these are:

- more complete standardised project repositories
- advanced method engineering
- MDA in a business centred architecture.

I have elaborated on two of these developments, MDA and method engineering, in chapters 8 and 9 respectively.

Sub question 6: What new developments can possibly lead to a more efficient development process in the future?

In chapter 7 I have tried to answer the question 'What new developments can possibly lead to a more efficient development process in the future?'

I have shortly discussed the following new developments:

- mature component based development
- advanced web services
- model driven architecture
- an unpredictable paradigm shift

More mature component based development is a possible development that depends on standardisation of interfaces, standardisation of documentation,

and awareness of what component based development is. This development would make software development more efficient, because software can be developed quicker, and the quality can be guaranteed better if the components are from a trusted source. There are no technical problems that hinder this development.

Like mature component based development, advanced web services is a development that does not depend heavily on technical developments, but mostly on organisational ones. This development could make software development more efficient for the same reasons as advanced component based development.

Automated method engineering might make the preparation for the software development easier and faster. A project manager would only have to describe the development situation to the method engineering system, and it would create a complete and usable software development method. The software development process itself might benefit if the automatically engineered methods would offer a better fitting method for each development situation.

Real method engineering or configuration does not seem to take place in practice yet. The book Method Engineering [HarmsBrink02] describes a theoretical framework for method engineering, but the example cases that are described in the book do not use this framework and describe relatively straightforward web pages that help present the method to the users.

The Method for Method Configuration (MMC) [KarlsÅger04] is not for the engineering of a complete method, but for configuring an existing method. The description of this method leaves a lot of questions and is not directly usable, but it is an interesting theory that might be of practical use in the future.

An application that generates complete and usable methods is not yet possible. The MMC could be a step towards this application, but MMC seems not to be usable in practice yet.

The fact should be considered that in software development, like in any field of work, developments occur that can not be predicted. We will probably not be surprised by developments that occur a year from now, but we can not accurately predict how software will be developed in twenty years.

10.2 The main question

In this section I will answer the main research question: 'How can improvement of tool support and development processes help make software development

more efficient?'

There are several problems that cause inefficiency in the software development process. According to software developers the most important causes for these problems are people related. However, people related causes for inefficiency are beyond the scope of this thesis. I have identified five causes for inefficiency that are people or tool related. Remedying these causes would make software development more efficient.

The activities from the software development that can be automated are coding and verification, and this can be done in the tool Statemate. However, in order to automate these activities, the activities that produce the required input have to be performed extra rigorously. So, the work has not disappeared altogether, it has shifted to other activities.

This thesis contains a theoretical framework for software development, which identifies of which activities the software development process consists. In my evaluation of which activities can be automated, I have only considered these activities. Other automatable activities might be found if a different theoretical framework for software development is used, or if the activities from the framework from this thesis are divided further in sub activities.

The software developers that I have interviewed had several ideas for changes in the way software development is done. Some of these ideas were wishes for change in the way software is developed, and some of them were just predictions of how software development will change in the future. I have researched these ideas, and have aggregated the following developments, than can help making software development more efficient:

- Mature component based development
- Advanced web services
- Advanced method engineering
- More complete standardised project repositories

These developments are partly tool related, and partly process related. The first two of these abilities do not especially depend on technical improvement, but more on standardisation and correct usage patterns.

In conclusion, improvement of tools and process can make software development more efficient, by pursuing the developments from the list above.

Carma McClure's prediction from the front page of this thesis, *'The only realistic way to increase software quality and productivity significantly is through automation,'* has not proven to be accurate. Automation can help improve software development, but it is not the only realistic way.

Appendix 1. Bibliography

Tools and methodologies

UML

[UML] http://www.uml.org [last checked: 06-jun-2003]
This is the official UML site of OMG.

MDA

[OMG03] Joaquin Miller and Jishnu Mukerji (editors), 2003
MDA Guide Version 1.0.1
http://www.omg.org/docs/omg/03-06-01.pdf
This document is a guide to MDA. A description of this document from the OMG site itself: *'This document is the most current description of MDA'* [last checked: 24 march 2004]

[MDACourse1] A course on MDA by University Twente
http://wwwhome.cs.utwente.nl/~aksit/courses/mda.htm

[MDACourse2] A course on MDA by Washington University in St. Louis
http://www.cait.wustl.edu/courses/OOP120.co

[MDACourse3] A course on MDA by InferData
http://www.inferdata.com/training/ooad/MDA/mdaoverview.html

[xmlCoverMDA] Article on MDA from Cover Pages
http://xml.coverpages.org/omg-mda.html

[Hazra] Tushar K. Hazra, MDA brings standards-based modeling to EAI teams
http://www.adtmag.com/article.asp?id=6311

Software development methods

[Kruchten00] The Rational Unified Process: An Introduction (second edition), Addison-Wesley, 2000. ISBN: 0201707101.
This is a popular and insightful introduction to the RUP.

[HarmsBrink02] Frank Harmsen and Sjaak Brinkkemper, Method Engineering - Web-enabling van methoden voor systeemontwikkeling, Ten Hagen Stam Uit-

gevers, 2002. ISBN: 9044003720

[KarlsÅger04] Fredrik Karlsson and Pär J. Ågerfalk, Method configuration: adapting to situational characteristics while creating reusable assets, Information and Software Technology, Volume 46, Issue 9, 1 July 2004, Pages 619-633 http://www.sciencedirect.com/science/article/B6V0B-4BHY5B2-1/2/b2e18197e92226aa9acca19da1cb3960 [last checked 24 August 2004]

[DSDM] The DSDM Consortium homepage. DSDM stands for 'Dynamic Systems Development Method'.
http://www.dsdm.org [last checked 31 August 2004]

[RUP] The homepage for the IBM Rational Unified Process.
http://www.ibm.com/software/awdtools/rup [last checked 7 September 2004]

[BriHarSae] S. Brinkkemper, M. Saeki, F. Harsens, Meta-modelling based assembly techniques for situational method engineering, Information Systems 24 (1999) 209 – 228

[Vens101] R. Vens, Referentiemodel voor een business-centered architectuur - Deel 1: Motivatie en basisprincipes, CIBIT, Utrecht, 2001

[Vens201] R. Vens, Referentiemodel voor een business-centered architectuur - Deel 2: Business domein component, CIBIT, Utrecht, 2001

Tools

[Arcstyler] Homepage for ArcStyler.
http://www.arcstyler.com/

[EnterpriseA] Homepage for Enterprise Architect.
http://www.sparxsystems.com.au/ea.htm

[SysArch] Homepage for System Architect.
http://www.popkin.com/products/system_architect.htm

[iUML] Homepage for iUML of Kennedy Carter.
http://www.kc.com/products/iuml/index.html

[Jython] Homepage for the scripting engine that uses the Python language runs on the Java virtual machine.
http://www.jython.org

[Aris] Site of IDS Scheer AG, the creators of ARIS Toolset.
http://www.ids-scheer.com/ARIS_Process_Platform/

[Select] Site of Select Business Solutions, the company that makes among other products Select Enterprise, Select Component Factory and Select Component Architect.
http://www.selectbs.com/

[SelectMDA] Site for the tool Select Solution for MDA
http://www.selectbs.com/products/products/select_solution_for_mda.htm

[JUnit] Homepage for the Java test framework JUnit
http://www.junit.org/

[MagicDraw] Homepage for the UML modelling tool Magic Draw
http://www.magicdraw.com/

[Statemate] Homepage for the specification and testing tool Statemate, which is aimed at embedded systems.
http://www.ilogix.com/statemate/statemate.cfm
http://www.ilogix.com/pdf/StatemateBrochure(V4).pdf contains the brochure.

[ReqPro] Homepage for the requirements tracing tool Requisite Pro from IBM.
http://www-306.ibm.com/software/awdtools/reqpro/

[ClearCase] Homepage for the Software Configuration Management tool Clear-Case.
http://www-306.ibm.com/software/awdtools/clearcase/index.html

[SoftwMod] Homepage for Rational Software Modeler.
http://www-306.ibm.com/software/awdtools/modeler/swmodeler/index.html

[Office] Homepage for Microsoft Office.
http://office.microsoft.com

[Erwin] Homepage for the database modelling tool AllFusion ERwin Data Modeler.
http://www3.ca.com/Solutions/Product.asp?ID=260

[Visio] Homepage for the drawing tool Microsoft Visio
http://office.microsoft.com/visio

[VisualStudio] Homepage for the development environment Visual Studio.
http://msdn.microsoft.com/vstudio/

[InVision] Homepage for the project management tool Project InVision
http://www.projectinvision.com/

[SoDA] Homepage for the reporting tool SoDA
http://www-306.ibm.com/software/awdtools/soda/features/

[Javadoc] Homepage for the documentation tool Javadoc
http://java.sun.com/j2se/javadoc/

[AndroMDA] Homepage for the tool AndroMDA
http://www.andromda.org/

[Aris] Homepage for Aris Toolset, the process modelling tool of IDS Scheer
http://www.ids-scheer.com/international/english/products/aris_design_platform/23244

Miscellaneous

[McClure89] Carma McClure – CASE is software automation, Prentice Hall1999. ISBN: 0-13-119330-9

American Scientists online – The Post-OOP Paradigm
http://www.americanscientist.org/Issues/Comsci03/03-03Hayes.html [last checked: 20 June 2003]

[Whittaker03] James A. Whittaker, Ph.D. and Jeffrey M. Voas, 50 Years of Software: Key Principles for Quality. Article in Software Quality Management Magazine.
http://www.srmmagazine.com/issues/2003-01/50years.html [last checked: 22 June 2003]
This article gives an overview of the last 50 years of software development, and gives recommendations to improve software quality.

[Sommerville95] Ian Sommerville - Software Engineering, fifth edition, Addison-Wesley, 1995. ISBN: 9-780201-427653.
This book gives information about many aspects of software engineering.

[Sommerville00] Ian Sommerville - Software Engineering, sixth edition, 2000. Addison-Wesley. ISBN: 020139815X

[Veldhuijzen van Zanten03] G.E. Veldhuijzen van Zanten, S.J.B.A. Hoppenbrouwers, H.A. Proper, System Development as a Rational Communicative Process, Department of Information and Knowledge Systems, Radboud University Nijmegen, Nijmegen, 2003.

[SEIDefinitions] Software Engineering Institute, What is Software Engineering http://www.sei.cmu.edu/about/overview/whatis.html [last checked: 4 juli 2003]
This is a list of definitions of software engineering from various sources.

[Bauer72] Bauer, F. L. Software Engineering. Information Processing 71., 1972

[Fairley85] Fairley, R. Software Engineering Concepts. McGraw-Hill, New York , 1985

[Alspaugh03] Alspaugh, Thomas A., What is requirements engineering? www.isr.uci.edu/~alspaugh/ics102.WhatIsRE.pdf [last checked: 23 juli 2003]
This is a nice and clear short set of slides about requirements engineering, made for a class.

[Nghiem03] Nghiem, Alex, Common mistakes in Adopting CBD. http://www.informit.com/articles/article.asp?p=31476 [lasted checked 17 march 2004]

[Kurtosuchi04] Brian T. Kurotsuchi, Design Patterns Tutorial
http://www.csc.calpoly.edu/~dbutler/tutorials/winter96/patterns/ [last checked march 23 2004]
Short, clear and unpretentious explanation of design patterns, with further references.

[CORBA] Object Management Group
http://www.omg.org/corba/
The startpage of the Common Object Request Broker Architecture.

[J2EE] Java 2 Enterprise Edition
http://java.sun.com/j2ee/
The startpage for this technology from Sun.

[Proper02] Erik Proper, Da Vinci - Architecture-driven Information Systems Development, Department of Information and Knowledge Systems, Radboud University Nijmegen, Nijmegen,2002.
Nijmegen Institute for Information and Computing Sciences, University of Nijmegen
Found at http://www.niii.kun.nl/~erikp [last checked 23 june 2004]

[Taula03] Anne Taulavuori, Eila Niemelä, Päivi Kallio, Component Documentation – A key issue in software product lines.
Information and Software Technology. Vol. 46 (2004) No: 8, 535 – 546

[wikiWebS] Wikipedia article on web services, which contains an introduction and useful web links.
http://en.wikipedia.org/wiki/Web_service

[MOF] Site for the Meta Object Facility from the OMG
http://www.omg.org/mof

[Hubert] Interview by the Code Generation Network with the Richard Hubert, CEO of Interactive Objects, which is the company that created ArcStyler.
http://www.codegeneration.net/tiki-read_article.php?articleId=12

[CMM] Homepage for the Capability Maturity Model.
http://www.sei.cmu.edu/cmm/

[Atkinson] Colin Atkinson - Unifying MDA and Knowledge Representation Technologies, University of Mannheim

[Boyatzis] Boyatzis, R.E. - The competent manager: A model for effective performance, John Wiley & Sons, New York, 1982