TECHNISCHE UNIVERSITEIT EINDHOVEN

Department of Mathematics and Computing Science

MASTER'S THESIS

An Agile Approach Supported by a Tool Environment for the Development of Software Components

Gertjan Zwartjes and Joost van Geffen
{g.zwartjes, j.v.geffen}@student.tue.nl

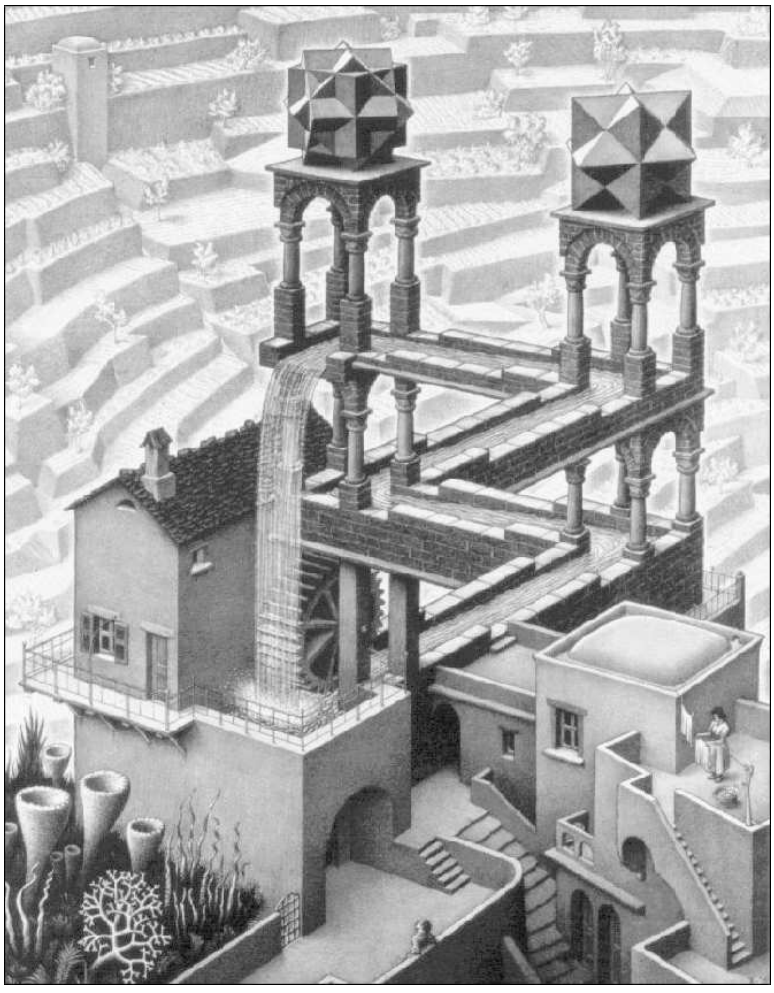
Supervisor: Prof. Prof. Dr. B.W. Watson [*+]
watson@win.tue.nl

External advisers: Prof. Dr. D. G. Kourie [+], Dr. A. Boake [+]
dkourie@cs.up.ac.za, andrew.boake@up.ac.za

[*] Technische Universiteit Eindhoven, Eindhoven, The Netherlands.
[+] University of Pretoria, Pretoria, South Africa

Eindhoven, January 2005

An Agile Approach Supported by a Tool Environment for the Development of Software Components



Gertjan Zwartjes and Joost van Geffen

4th January 2005

Colophon

This document is typeset with LaTeX 2e, TeX version 3.14159, Web2C 7.4.5 in 11pt Times, Helvetica and Courier. The following LaTeX packages are used: pslatex, babel, graphicx, geometry, fancyhdr, url, setspace, acronym, ifthen, fancyvrb, tocbibind and rotating. The bibliography is maintained with BibTeX.

Figures 2.1 and 4.1 are created with MetaPost. The picture on the title page is M.C. Escher's "Waterfall", Copyright © 2004 The M.C. Escher Company – Baarn. Used by permission. All other figures are created with OpenOffice.org Calc.

Abstract

A rigorous software engineering methodology, applied in the development of software components, causes overhead costs, for example, by over-documenting. The alternative is to use an agile approach, to reduce overhead, caused by process-centered tasks. The advantages and disadvantages of rigorous and agile approaches are analyzed, and criteria to choose between them are introduced. As a solution, this dissertation proposes an agile process for the development of software components.

The importance of tools and their relation with a process are discussed. Software development tools may contribute greatly to the success of a project, and may provide critical support for a software engineering process. Therefore the selected set of tools in a development process is essential. The selection process must be transparent and carried out with care. An approach that may be followed to select a tool environment is proposed. The results of case studies, where the approach is retrospectively applied, confirm the importance of tool selection.

A set of tools is selected for the tool environment to support the component development process. It is not always the case that the needed tools can be selected out of an existing set. The practical work that was needed to custom develop two tools — an automated build system and code documenting tool — is described. The results of applying the proposed process to several projects are discussed, as well as interviews with the responsible developers. Evidence is presented, showing that both the proposed process and the tool environment chosen, reduce project overhead and cost.

Samenvatting

Een rigoreuze methodologie voor software engineering toegepast in de ontwikkeling van herbruikbare (software) componenten resulteert in overhead kosten, bijvoorbeeld door onnodige documentatie. Het alternatief is een agile aanpak om de overhead kosten te bestrijden die veroorzaakt worden door process-georiënteerde taken. De voor- en nadelen van de toepassing van een lichtgewicht aanpak worden vergeleken met een rigoreuze aanpak. Ook worden de criteria om een keuze tussen de twee verschillende manieren van aanpak te maken beschouwd. Als oplossing voor de ontwikkeling van de componenten wordt in deze scriptie een op maat gesneden agile proces gegeven.

Ook wordt belicht hoe belangrijk tools, oftewel gereedschappen, zijn in het ontwikkelproces. De precieze relatie tussen gereedschappen en het proces wordt onderzocht en de conclusie is dat gereedschappen een grote invloed hebben op het succes van een project; de gereedschappen kunnen in belangrijke mate bepalend zijn in het ondersteunen van een bepaald proces. Dit heeft als gevolg dat het maken van een selectie van gereedschappen dermate belangrijk is, dat het selectieproces transparant en zorgvuldig moet worden uitgevoerd. Een aanpak wordt voorgesteld om een verzameling van gereedschappen te kiezen. Een aantal case studies waar de aanpak achteraf wordt toegepast bevestigen de cruciale rol van de selectie van het gereedschap.

Een verzameling tools wordt gekozen met de beschreven aanpak voor het component ontwikkelproces. Het is niet altijd het geval dat de benodigde gereedschappen gekozen kunnen worden uit een bestaande verzameling. Twee gereedschappen, een automatisch build systeem en een code documentatie systeem, zijn op maat ontwikkeld en deze gereedschappen en de aanpak van het praktische werk wordt in deze scriptie beschreven. De resultaten van de toepassing van het proces en de gekozen en ontwikkelde tools tijdens een aantal projecten worden beschreven, onder andere door middel van interviews met de verantwoordelijke ontwikkelaars. Er wordt aangetoond dat zowel het component ontwikkelproces als de gekozen gereedschappen tot een vermindering van overhead leidt.

Table of Contents

List of Figures

List of Tables

Preface

This document describes the work for our master's thesis, from November '03 until November '04. After having worked together successfully in the past for many times, we decided that it would be a great idea to combine our skills and shared interests in this project again. We were already working for the same employer, who was also interested in taking his chance with us in a graduation project. Our supervisor approved the project and the work started early November '03.

The work for our employer consists of the development of software components, that will be described in more detail in Chapter 1. The first process, that we used to manage the development of these components, was the European Space Agency (ESA) Software Engineering Standard. This process, however, was too heavyweight for the specific context. It was decided to switch to the ESA lite standard, but a significant amount of overhead was still caused, by writing documents, with content that was felt not optimally useful.

For our employer, a goal of the research was to find an optimal process for the development of the components. The first challenge at the start of the graduation project was to broaden our scope of software engineering. We studied the literature on the topic and the result is presented in Chapter 2.

From mid February '04 to mid April '04 we went to the University of Pretoria in South Africa. Throughout our stay, we discussed the importance of tools in a development process with Andrew Boake and Derrick Kourie, who became our external advisers. The process of tool selection was found to be very interesting and it was decided to write down our thoughts. It turned out to be a very interesting subject and the resulting text was written in the form of an article that was submitted to the South African Institute of Electrical Engineers (SAIEE), for a special issue on software engineering. After a review and a rewrite, it was accepted and the paper will be published in the December '04 issue of SAIEE.

The paper is included as Chapter 5, with minor adjustments to have it fit in this dissertation. We used the technique to validate the tools that were chosen to improve the component development process. When we returned from South Africa, Chapter 3 was written, containing the background material that was used to write the paper for SAIEE.

Broadening our software engineering scope, resulted in starting to strip the ESA lite process to allow it to be more lightweight. In the end, after using the process in practice, we concluded that the newly formulated process was suitable in the specific component development context. The process description is included in Chapter 4. A document is derived from this chapter, that prescribes the process standard for the development team that is responsible for developing the components. Chapter 4 has a high level of detail, because it was written to serve as a base for a standalone process description document.

A significant part of our graduation year consisted of practical work for our employer. He hired us to improve the quality of his project, by implementing software components to replace and add features to his product. We implemented several components and a support application, to test out the component development process. Additionally, we implemented an automated build and release system and a code documenting system to streamline the component development process further. For the support application and code documenting system the component development process was tested whether it scales to applications. These case studies can be found in Chapter 6.

The majority of the research described in this dissertation is a collaborate effort of both the authors. This document is *pair-written* in a way similar to the *pair-programming* technique of Extreme Programming. Each part that was not collaboratively written, was extensively reviewed by the other author. The practical work is, however, more or less, distributed; each case study has a responsible lead-developer. For the collections component, the build system, and API documentation system, Zwartjes is the lead-developer. For the internationalization component and support application, van Geffen is the lead-developer.

Acknowledgments

We would like to thank a number of people[1] that contributed in many different ways to the research in this dissertation. First of all, we are indebted to Bruce Watson, our supervisor who guided us through the process of writing this dissertation. Despite his busy schedule, he was able find the time to warmly welcome us into his office many times.

We would especially like to thank Derrick Kourie, Andrew Boake and everybody from the Polelo lab at the University of Pretoria for their hospitality. In South Africa, Derrick Kourie and Andrew Boake were a catalyst for us to start working and they made us feel at home. We very much enjoyed working together. As a highlight of our time in South Africa, we will never forget our trip to Kruger with Richard and Lezel.

Funds for our research in South Africa were provided by the *TU/e Fonds Studiepunten Buiten Nederland (FSBN)* and the *Koninklijk Instituut Van Ingenieurs (KIVI)*.

We would also like to express our thanks to our colleagues at Intersoft Software Research, Wouter Bijlsma and Rick van Bijnen, for their support. They never avoided lively discussing the topics in this dissertation, reviewed several draft versions, and contributed to the practical work, for which we are very grateful. We would also like to thank our colleagues at Intersoft Software Engineering, Misja IJzerman, Bart Bakker, Milo Visser, Rene Pijnacker, Jeroen Schouten and Jurjen Pieters, for their unending practical constructive criticism. Walter Lagerweij helped us with our office paperwork.

Among other reviewers, the following people deserve credit for helping us polishing this dissertation: Derrick Kourie, Morkel Theunissen, Tom Verhoeff and our exam committee members Andrew Boake and Alex Telea. In addition, we received valuable feedback from Cees Hemerik, after giving a presentation of our work.

We are immensely grateful to Richard Kloosterman for his faith in us and providing the resources to do our research. Working together with Richard has always been a pleasure for both of us. His ideas and views on bringing theory in practice contributed greatly in maturing our work.

Personally Gertjan would like to thank his parents, Helma and Gerrie, and his girlfriend, Marieke, for their unconditional support during his years at the university and especially this last year. Joost would like to thank his parents for their support and making it financially possible to attend university. He also wants to express his gratitude to Lisenka who supported him in his decisions and helped him at difficult moments.

[1] Titles are omitted.

Chapter 1

Introduction

In 1991, our employer, Intersoft, set out to produce an enterprise administration system, targeted at small to medium sized companies. The development team was relatively inexperienced and had little formal software engineering background. Nevertheless, they persevered, albeit in a somewhat unstructured manner, successfully developing and marketing the system to a client base of approximately 300 installations. Unsurprisingly, the system evolved over time in response to client and market needs. After some 10 years, the team realized that the application had become increasingly unmanageable, and something needed to be done for work to continue.

The authors were hired as consultants to augment the efforts of the existing developers. The steps were taken to overall re-engineer and restructure the software into manageable components. Intersoft is located in Amsterdam, however, practical considerations require that we work in a remotely located office in Eindhoven. The part of the organization in Eindhoven is called Intersoft Software Research. Face-to-face meetings with the existing developers are held on a regular base. Our primary task is to implement software components to replace and add features of the administrative application. The development of each component therefore requires close collaboration between the two parties. The term project is used to denote the entire process of the development of a component.

This dissertation describes the reseach that has been conducted to improve the development process of the software components. Therefore, two fields are investigated: (1) software engineering methodology, and (2) software development tools.

Software engineering has been a research topic for almost four decades. Software engineering methodology[1] originated in the 1960s, producing heavyweight methods. More recently, a leightweight form of methodology has emerged. Software development tools have been used since the emergence of software development itself, and are frequently deployed to improve the development of software.

The relevance of the subject can be illustrated by the roadmap to software engineering tools and environments, published in April 2000, by Ossher, Harrison, and Tarr. They describe several major trends that will emerge in the future:

[1] Methodology formally refers to the science or study of methods. However, the term is frequently (ab)used to denote a body of methods, rules, and postulates. In software engineering, the term methodology is commonly used to refer to a set of recommended practices.

> [We expect to see] new methodologies, formalisms, and processes to address non-traditional software lifecycles, and the tool and environment support to facilitate them. With significant economic, business and research pressures to be first to market or first to publish, especially in new domains like pervasive computing and e-commerce, fewer individuals are able to expend the resources on a traditional spiral lifecycle, even knowing what they are losing as a result of cutting corners. Further, with rapid development, rapidly changing requirements and platforms, and an extremely fast-moving and volatile field, it is questionable whether many new domains would actually benefit from the more costly traditional software lifecycle — it simply takes too long. Current methodologies and tools generally do not work well in such non-traditional contexts, and they do not "degrade" gracefully. Developing methodologies and models that do work in these contexts, and tool and environment support for them, is a critical challenge of the upcoming decade. [Ossher *et al.*, 2000]

In this dissertation the expectations from Ossher, Harrison and Tarr are tried to be fulfilled by: (1) proposing a process for the development of software components, and (2) providing the tool environment for the process. In addition, an approach to select the tools for a tool environment is described as well.

To understand the origin and background of the proposed process, the history and relevance of heavyweight and agile methodologies are described. Both methodologies will be compared and a short introduction is given into choosing a methodology. The background information will be used to explain the important matters of the component development process. In addition to heavyweight and agile methodologies, the open source movement will be discussed as well. The open source movement is one of the latest increasingly popular group of people developing software. It will provide an example where the development process depends on both the process and the use of tools. The open source movement also influenced work on process for the development of software components and the supporting tool environment.

After software engineering methodologies are introduced, the impact of tools and processes on software development in recent years is investigated, including the latest developments in software engineering methodology, and the effect of these developments on tools. Although tools are certainly not a *Silver Bullet* [Brooks Jr., 1987], they are perhaps more important than generally realized. The open source community and the tools used in the process of developing open source software, serve as an example for this relation.

The conclusion that tools are important in the development process logically implies that tool selection or, in case no existing tools are suitable, tool customization is an important influence on the success of a project as well. An approach for managing the selection process is outlined, that visualizes the process of selection in a matrix. Tools are analyzed in the approach by specifying functional requirements and supported properties.

The tools were already chosen for the development process, before the tool matrix had evolved. However, the model is applied to validate the set of tools that was chosen. Especially in the cases where it was decided to custom develop a tool, the model is used to argument the reason to custom develop the tool. Finally, the practical work concerning the implementation of several components and a support application is discussed. These case studies present the results of applying the component development process and tool environment in practice.

1.1 Research Questions

The main theme of this dissertation is to find means — in the form of a software engineering process and a tool environment — to enhance the development of the components, described in the previous section. To solve this problem, a number of research questions are formulated, that will be addressed in the remainder of the dissertation. In Chapter 7, a summary of the results is presented, according to the research questions in this section. The following list enumerates the ten main research questions:

1. *What is the current scope of software engineering methodology, and more specific, what are the main trends or directions in the area?* To enhance the development of the components, by introducing a software engineering process, it is important to first explore the options. By looking at software engineering methodologies the highest level of options is examined.
2. *What are the differences between the directions of software engineering methodologies, and can software engineering methods be classified?* To choose between the high level options it is important to list the differences between directions in software engineering methodologies, and, if possible, specify the means to classify software engineering methods.
3. *What are the main considerations that influence the choice of a software engineering methodology for a project?* If there are grounds or criteria to choose between software engineering methodologies, they can be applied to argument and answer the next question.
4. *What software engineering methodology is best suited for the development of software components?* With this question, we try to discover the type of software engineering methodology, best suited for the component development process. This answer may be the base to analyze the next question.
5. *What software engineering method is best suited for the development of software components?* The answers to the previous questions are used to determine the choice for a software engineering method for the component development process.
6. *What impact do tools have on a project or, more general, what is their impact on a development process?* The goal of this question is to find possibilities to enhance a development process by introducing certain tools — to determine the influence of tools on a software engineering method.
7. *How to choose the tools for a development process, or more specific for a project?* Because many tools of the same genre exist, it is difficult to find that one tool that fits a developers need best. Can the decision process be generalized to select a set of tools?
8. *Which tools are commonly used to enhance the development process?* Is there a common set of tools that is frequently used in projects to improve the development process?
9. *Which tools are important to support the process for the development of software components?* We try to find a set of tools, that enhances the process that was found to be best suited for the development of the components, from question number 5.
10. *What is the relation between people and a software engineering process?* In what way do people affect the development process that is applied in a project, and in what way are people influenced by a development process in a project?

1.2 Overview of the Document

Chapter 2 reviews the history and facts about software engineering methodologies. It will bring up details about the open source software development model and rigorous and agile software engineering. Chapter 3 discusses the importance of tools in the development process. In Chapter 4, a customized software development process for the development of software components is illustrated. To support this process, tools will be chosen using an abstract model discussed in Chapter 5. Chapter 6 elaborates how an Application Programming Interface (API) documentation system and a build system have been developed for the tool environment of the development process, as well as three case studies that apply the proposed process from Chapter 4. Finally the conclusions are presented in Chapter 7.

Chapter 2

An Introduction to Software Engineering Methodologies

The sooner you start coding your program, the longer it is going to take. Murphy's law of programming [Formulated by H. Ledgard, 1975.]

It is easier to change the specification to fit the program than vice versa. Alan J. Perlis: Epigrams in Programming SIGPLAN Sept. 1982

In this chapter, general aspects of software engineering methodologies and processes are described. Three common methodologies — heavyweight (or traditional) software engineering, lightweight (or agile) software engineering, and the open source software development model — are outlined in more detail. The important properties of the methodologies, their history and development over time are highlighted. In addition, our experience with each of the methodologies is included.

In Chapter 4, a software engineering process will be derived based on the methodologies described in this chapter. A summary on how to choose among the software engineering methodologies is included in Section 2.7.

2.1 Introduction

The term *software engineering* was first used around 1960 as researchers, management, and practitioners tried to improve software development practice. The NATO Science Committee sponsored two conferences on software engineering in 1968 (Garmisch, Germany) [Naur and Randell, 7 11 October 1968] and 1969 [Randell and Buxton, 27 31 October 1969], which gave the field its initial boost. Many believe these conferences marked the official start of the profession. Programmers

were well aware of electrical and computer engineering, and debated what software engineering might mean. The software engineering area was stimulated by the *software crisis* of the 1960s, 1970s, and 1980s, when many software projects had bad endings. The software crisis was the name that linked together the many problems with software.

In the early 1980s, development of software for the US Military was chaotic. Most projects were late, over budget and deficient in functionality. Some even caused property damage or loss of life. The Software Engineering Institute (SEI), an entity attached to the Computer Science Department of the Carnegie Mellon University in Pittsburgh, was established in 1984 to advance the practice of software engineering in the USA. The SEI has been the origin of many groundbreaking initiatives in software engineering, investigated, tested, and applied in collaboration with its partners. An example is the Capability Maturity Model (CMM), which positions a software development organization at a designated level of maturity, based on its development processes and management practices. Another example is its leading work in identifying, specifying and encouraging ongoing adherence to qualities of a good system architecture.

Since that time, the SEI has produced much work which forms valuable input for any software development effort that wants to improve its delivery record. Its mission is "to transform software engineering from an ad-hoc, labor-intensive activity to a managed, technology-supported engineering discipline so that the government can acquire systems from a broader, more capable contractor base." The SEI continues to do research aimed at solving significant software industry problems, coming up with innovative solutions in collaboration with its partners (including university departments), testing these solutions and disseminating the knowledge gained widely. The SEI has contributed largely to the improved state of software development in the USA.

In 1987, Brooks published the famous paper *No Silver Bullet* [Brooks Jr., 1987]. In this article he argues that no individual technology or practice can make a 10-fold improvement in productivity in 10 years time. Almost every new technology or practice that was introduced claimed to be the solution for this software crisis. Object Oriented Programming (OOP), methodologies, processes, the Unified Modeling Language (UML), and CMM are examples of technologies and practices that were supposed to be a silver bullet solving the software crisis. None of them actually turned out to be silver bullets, however, such practices can make incremental improvements in productivity and quality.

Following a software engineering method or standard is a practice that can improve productivity and quality. Software engineering methods span many disciplines, including project management, analysis, specification, design, coding, testing, configuration management and quality assurance. These methods can informally be classified into heavyweight and agile (lightweight) methods. Heavyweight and agile refer to the amount in which the disciplines mentioned above are elaborated in a method. The next sections examine these methodologies in more detail.

2.2 Heavyweight Software Engineering

Heavyweight methods include a large amount of formal process, paperwork and documentation. In a heavyweight methodology strict attention to rules and procedures is demanded and deviation from the standard is strongly discouraged. Regardless what kind heavyweight method is applied within a software engineering project, a specific attitude must be adopted towards working in a process-oriented environment in which planning and documentation is emphasized. The most

important characteristics of a heavyweight methodology, identified by Khan [Khan, 2004], are:

- **Plan oriented** – Heavyweight methods try to capture all possibilities in advance. An exact planning is made at the start of a project in which each phase is elaborated in detail. When a specific planning of a phase in the project has not been fulfilled, changes have to be made in the planning for the remainder of the project. Project quality is measured by conformance to plan. The most important factors to consider a project a success, is on-time and on-budget delivery.
- **Comprehensive documentation** – Comprehensive documentation provides a thorough insight to the innards of the project at anytime and is critical to the success of a project. In addition, it is a prerequisite for future maintenance.
- **Predictive approach** – Software development is approached as a predictive and repeatable activity along the line of the engineering disciplines. This predictive approach is closely related to the fact that heavyweight methods are plan oriented.
- **Process oriented** – A heavyweight methodology focuses on developing a process in which different roles are identified, each having clearly defined tasks.
- **Big design upfront** – Heavyweight methodologies are based on comprehensive requirements gathering upfront. The goal is to build a scalable and flexible architecture. The architectural design is an eye-opener into the complexity involved in the project.

Heavyweight methodologies are applied within large and more critical projects. Cockburn writes about selecting the methodology for a project: "A relatively small increase in methodology size or density adds a relatively large amount to the project cost" [Cockburn, 2000]. A direct consequence is that a heavyweight methodology should only be applied when the extra costs involved with this approach are inevitable. And this is exactly the case in large and critical projects. Examples of heavyweight software engineering methods are Rational Unified Process (RUP), Cleanroom, Fusion, and the ESA standard.

Because heavyweight software engineering methodology is more or less defined by the actual heavyweight methods, this chapter will describe the heavyweight methodology by example. In the next section, Fusion will be elaborated in more detail to show how a heavyweight method is implemented in practice.

The section following Fusion, illustrates the concept of heavyweight methodologies by means of the ESA software engineering standard. The ESA software engineering standard is described in great detail because: (1) the process introduced in Chapter 4 is based on the light version of the ESA standard [European Space Agency, 1996], and (2) it is the first heavyweight standard we worked with. We gained a lot of experience working with the ESA standard. These experiences are described in a separate section as well.

2.3 Fusion

Fusion was developed to provide a systematic approach to object-oriented software development. This is described in Coleman's book about the Fusion method [Coleman *et al.*, 1994]. It integrates and extends approaches that existed at the time of the development of Fusion. It is a method that

provides for analysis, design and implementation. Fusion tries to give a direct path from the requirements of the system to an implementation in some programming language. This development method supports both the technical and managerial aspects of software development.

Fusion divides the software development process into phases and gives clear indications on what has to be done in each phase. Fusion provides criteria that tell the developer when to proceed to the next phase. In addition management tools are provided for software development. There are three phases in the Fusion process: (1) the analysis phase, (2) the design phase, and (3) the implementation phase. Note that Fusion does *not* have a requirements phase, because requirements capture is usually performed by a customer and it is his job to deliver the initial requirements document. The output of each phase is clearly identified and cross-checks are defined to ensure consistency between and within phases.

In the analysis phase, the intended behavior of the system is defined and models that reflect this behavior are produced. The models describe the classes of objects in the system and the relations that exist between those classes. The operations that can be performed on the system are classified together with the allowable sequences of those operations. In contrast to some other standards, no methods are attached to particular classes in this phase. In the Fusion method this practice is done in the design phase. The designer in the design phase chooses an implementation for the system operations by the run-time behavior of interacting objects. Operations and attributes are attached to the classes and the appropriate relationships between classes are formulated. In the implementation phase, the models constructed in the design phase are turned into code in a particular programming language.

Fusion is a method that can easily be adapted for different projects. The original Fusion is by nature a heavyweight software engineering process. A more lightweight version can be used in projects that cannot afford the additional effort required to use the full version. Parts of the Fusion process or notations can be used to complement weak points of other development processes.

2.4 The ESA Software Engineering Standard

The ESA is one of the originators of well-documented software engineering standards. On their website [Jones *et al.*, 1997] they describe in detail what made them develop their ESA Software Engineering Standard. In the next section their motivations are summarized. This historical background is followed by a section containing our own experience with the ESA software engineering standard.

2.4.1 Historical Background

The website describes that the ESA Software Engineering Standards originate around 1975. At that time the ESA was working on a number of ambitious projects involving the development of large amounts of code. Some of those projects were on relatively new subjects. Project groups often consisted of excellent engineers, but most of them were not used to working in projects with huge costs and a strict schedule. There was very little project discipline.

One of those projects involved a software development activity to support a mission of which the launch was very important. The project was late and there were some major hardware and

operating system problems. The software developers were waiting for improvements of the hardware. To be able to continue the work, they wanted to reduce the amount of requirements to only those highly essential for the launch of the satellite. But the problem was that nobody knew what the exact requirements were: "There were no written requirements — at best, some could partly be retrieved from minutes of meetings. All the rest were in the minds of the project engineers. How could cost and schedule be guaranteed under these circumstances? This was a project manager's worst nightmare!" [Jones *et al.*, 1997].

At this moment it was decided to stop any further development. All software engineers first had to write down their understanding of the requirements. Management was very skeptical about this decision, especially since the project was already late. However, as it turned out, it seemed to be working. As they had a written list of requirements, they were now able to select only those requirements needed for the launch. The remaining requirements were implemented after the satellite was launched. In this case a disaster was prevented from happening.

The Board for Software Standardization and Control (BSSC) was founded because of one very good reason: "We never want to find ourselves in this situation again!". This is the seed from which the BSSC, and consequently, the ESA Software Engineering Standards grew.

> The goal of the BSSC is to define a methodology that would enable:
>
> 1. The development of a product based on requirements defined by the users of the system, and not solely on the developers' wishes.
> 2. A rigorous testing of the system before its release for operation.
> 3. The execution of a project according to tight cost constraints and rigorous schedule control. In this regard, it must be noted that launch delays are extremely expensive and it is simply unacceptable to delay a launch because of delays in the development of software components, whether these are on board the spacecraft or in the ground system. [Jones *et al.*, 1997]

The first realistic task of the BSSC was to define the software life cycle and its phases. This resulted in initial guidelines for the development of software, issued at the beginning of 1978. Together with these guidelines, a common terminology was introduced. The next step was to produce a set of documents, each of which covers a specific phase of the software life cycle. Those documents describe milestones and the applicable practices for each phase. A lot of useful information was derived from other software engineering standards, especially from the Institute of Electrical and Electronic Engineers (IEEE). The BSSC has written a single document that describes how the documentation is organized [European Space Agency, 1991]. In addition, a general document [European Space Agency, 1995a] was created on how the standard works and how it should be applied in a specific project. Furthermore a document is written for each phase of the standard, in chronological order the user requirements definition phase [European Space Agency, 1995b], software requirements definition phase [European Space Agency, 1995c], architectural design phase [European Space Agency, 1995d], detailed design phase [European Space Agency, 1995e], transfer phase [European Space Agency, 1995f] and the operations and maintenance phase [European Space Agency, 1995g]. Project documentation is described in four documents, software project management [European Space Agency, 1995h], software configuration management [European Space Agency, 1995i], software verification and validation control [European Space Agency, 1995j] and software quality assurance [European Space Agency, 1995k].

"Although [the application of the ESA software engineering standards] in large projects is quite straightforward, experience has shown that a simplified approach is appropriate for small software projects" [European Space Agency, 1996]. The ESA introduced a special guide that provides guidelines on how to apply the standard to smaller projects. This is basically a simplified approach of the full ESA standard. In this guide a number of strategies suitable for small projects producing non-critical software are described, along with possible ways of tailoring the mandatory requirements to these small projects. This includes the possibility of combining and simplifying the various management documents. The ESA defines a project to be "small", if one or more of the following applies:

1. Less than two man years of development effort.
2. A single development team of five people or fewer.
3. Fewer than approximately 10000 lines of source code (excluding comments).

The ESA software engineering guide [European Space Agency, 1991], defines several criteria that influence how the standard is to be applied:

- The number of people required to develop, operate and maintain the software.
- The number of potential users of the software.
- The amount of software that has to be produced.
- The criticality of the software, as measured by the consequences of its failure.
- The complexity of the software, as measured by the number of interfaces or a similar metric.
- The completeness and stability of the user requirements.
- The risk values included with the user requirements.

The ESA qualifies a project of two man-years or less as a small one; projects of twenty man-years or more are qualified as big. The criticality of the software is considered a highly important factor when deciding how to apply the standards. When for example software — developed for the launch of an expensive rocket — fails, it can result in loss of life and money. So this kind of critical software has to be developed with care. It is up to the management or senior management of a project to determine in which amount these factors influence the actual project. They often have the final word in what standard is to be used and how it is to be applied. In 1994, the ESA had decided to start the implementation of a new internal system of standards.

> In June 1994, a resolution has been adopted by the ESA Council which confirmed the commitment of the agency to transfer the present Procedures, Standards and Specifications (PSS) system of ESA space standards to a new system of standards under preparation by the European Cooperation for Space Standardization (ECSS). Information and announcements about ECSS can be found on the official website, see [European Space Agency, 2004]. The ESA, European national space agencies and European space industry are represented in the ECSS. The PSS system of standards is a mandatory input to the preparation of the ECSS standards. The ECSS system of standards covers management, product assurance and engineering standards, specifically for space projects. In ECSS, software engineering standards form a branch of the

> engineering standards. The ECSS standard for software engineering, ECSS-E40, is in preparation. ECSS-E40 will be a high-level requirements-oriented standard. Unlike PSS-050 [European Space Agency, 1991], it reflects the specifics of space projects, for example the relationship between the space system development cycle and the software development cycle. [Jones *et al.*, 1997]

The ESA Council has decided that no new issues of PSS documents will be released. The existing PSS standards will however remain in force until the relevant ECSS standards are available and adopted. A plan will be prepared for the introduction of ECSS-E40. It is anticipated that this will include tests using the ECSS-E40 standard in selected ESA projects. A detailed standard, practice-oriented, at the level of, for example, the current ESA Software Engineering Standards, would be expected as a requirement for the implementation of ECSS-E40. It would also be expected that other organizations (e.g. national space agencies, space industry) might use their own standards as implementations of ECSS-E40.

Within the agency there will be a continuing need for a PSS-050 type standard for much of the software development in the agency which is not directly part of a space project, e.g. ground infrastructure software, administrative software and technology studies.

2.4.2 Experience with ESA

Our first time experience with the ESA software engineering standard was during a software engineering project at the Technische Universiteit Eindhoven (TU/e) in 2000–2001. The development team consisted of 10 members. Each team member had to work on the project for approximately two man-months, this adds to a total of less than two man-years. The assignment was to develop the software using the ESA software engineering standard. We attended classes in which the basics of this standard were presented. One of the goals of this software engineering project was to get familiar with working according to a software engineering standard.

In the case of our software engineering project, senior management decided to use the full ESA software engineering standard, without any weakening. The development team itself had no saying whatsoever on how the standard was to be applied. It would have been better to use the ESA lite for this project, because the development effort was less than 2 months and the number of lines of code did not exceed 10.000. Only the number of project members was too high to use ESA lite. The project was also suited for an agile approach.

In that project we experienced that the ESA standard was too heavy and caused unnecessary overhead. We were focusing too much on the process, and tasks that were purely related to the process. One example was extensive management and product documentation. Furthermore, the strictly defined roles were unnecessary for the small sized and co-located development team. However, one of the goals of the course was for the development teams to gain experience working with a heavyweight standard. From this point of view, the educational aspect made the project valuable.

Each phase ended with the delivery of one or more documents that had to be reviewed extensively before its acceptance. As a result of delays in the early phases in which a lot had to be documented and reviewed, we found ourselves short in time for implementation and testing. Adding the difficulties of maintaining documentation in the implementation phase and the lack of time, more than 50% of the documents were found to be outdated by then end of the project. Besides the learning experience, it is questionable whether writing these documents was worth the

effort for the project's end-product.

2.5 Agile Software Engineering

Agile software development tries to give an answer to the business community asking for more lightweight development processes. Key elements of an agile development process are simplicity and speed. In a review and analysis of agile software engineering methods by Abrahamsson and associates, a method is defined to be agile when the software development is [Abrahamsson *et al.*, 2002]:

1. **Incremental** – small software releases and rapid cycles.
2. **Cooperative** – the customer and developers working constantly together with a lot of close communication.
3. **Straightforward** – the method itself is easy to learn and modify.
4. **Adaptive** – enables last moment changes.

It is accepted, especially within the agile community, that agile denotes a way of thinking and is not necessarily a fixed method. In software engineering, agile processes are low-overhead processes that accept that software is difficult to control. They minimize risk by ensuring that software engineers focus on smaller units of work, or focus on fewer, rather than too many, things at a time. In general, agile processes impose as little overhead as possible in the form of rationale, justification, documentation, reporting, meetings, and permission. Replacing before-the-fact permissions with after-the-fact forgiveness is one of the key elements of reducing overhead. The key principles of agile methodologies, identified by Khan [Khan, 2004], are:

- **People oriented** – People, which are customers, developers and end-users as well, are considered the most important factor of software development.
- **Adaptive** – Proponents of agile methodologies believe — and thus assume — that change is inevitable. People do change their mind, whether or not they signed a contract. That is why agile methodologies welcome change at every stage of a project. Adaptability means being able to quickly respond to changing conditions. This implies that creative people are preferred to be involved in the project.
- **Decentralized approach** – The decision-making process is decentralized. This means that the people that are actually writing the code can make the final decisions. This of course does not mean that technical people can take the role of the management people. Management is still responsible for general decisions and management activities. However, technical expertise of the developers has to be recognized by management. The chain of command — defined as the number of persons between the customer and the developers — should be minimal.
- **Limited size** – Small teams are preferred when an agile methodology is applied. Occasionally agile projects have succeeded with somewhat bigger teams, but in general agility does not work with big development teams.

The principles mentioned above can also be seen in table 2.1 that shows differences between agile and heavyweight methodologies. The agile methodology will be explained by its historical background and the *Agile Manifesto*. The agile movement defines a manifesto, unlike the heavyweight methodology, to identify processes as agile. Therefore, the Agile Manifesto will be explained in detail, but no detailed examples of actual agile methods are given.

2.5.1 Historical Background

Interest in agile methodologies has blossomed in the last couple of years. The roots however go back to the early 1990s. Agile processes evolved as part of the reaction against the existing heavyweight processes. These processes were seen as bureaucratic, slow, demeaning, and most important contradicted the way that software engineers actually work. This has been the origin for a lot of discussions between followers of agile and heavyweight software engineering methodologies, see for example the two great methodologies debates [Highsmith *et al.*, 2001, Highsmith *et al.*, 2002] that have been published in the Cutter IT Journal.

Exactly following a plan was no longer the most important goal of a project. Satisfying your customers at the time of delivery — and not at the start of a project — became more important. It seemed to happen that major changes in the requirements, scope, and technology occurred during the life span of a project. These changes were often out of control of the development team. Traditional approaches assumed that, by just trying hard enough, it would be possible to create the set of requirements early in advance and thus eliminate the cost of change. Nowadays, eliminating changes early means being unresponsive to actual business changes. It is, however, important to be careful to retain quality when embracing change. The market demands and expects innovative software of high quality meeting their needs, in as little time as possible. Agile methods are a response to this growing expectation.

2.5.2 Agile Manifesto

The term agile was born at a meeting of seventeen people who got together for three days, starting on February 11th 2001 in The Lodge at Snowbird ski resort in the Wasatch mountains of Utah [Beck *et al.*, 2001, Theunissen *et al.*, 2003]. Among those people were representatives of Extreme Programming (XP), Crystal, Feature Driven Development (FDD), SCRUM, Adaptive Software Development (ASD), Dynamic Systems Development Methodology (DSDM) and Pragmatic Programming. All these people felt the urgent need for a new approach to software development, rejecting the traditional heavyweight document-driven methodologies. Because of the opposite nature of these new methodologies, they were called ‘lightweight’. The goal of the attendees of the meeting was to find common ground between this diversity of lightweight methodologies. The result of the meeting was more than the attendees expected: not only a common basis for all the lightweight methodologies was discovered, it was considered extensive enough to form an alliance. The common basis they agreed on was laid down in a manifesto that was signed by all attendees. They agreed on the term ‘agile’ to classify the lightweight methodologies. The *Manifesto for Agile Software Development* was formulated as follows:

> We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

> Individuals and interactions over processes and tools
> Working software over comprehensive documentation
> Customer collaboration over contract negotiation
> Responding to change over following a plan
>
> That is, while there is value in the items on the right, we value the items on the left more. [Beck *et al.*, 2001]

Members that sign the Agile Manifesto adhere to the following principles:

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity — the art of maximizing the amount of work not done — is essential.
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

From these principles it clearly emerges that there are two basic important concepts: working code and people working effectively together, see [Highsmith and Cockburn, 2001a, Highsmith and Cockburn, 2001b]. Working code is a measurement for the developers and customers of what has already been done and more important what still has to be done to meet the expectations of the customer. No matter whether the code will be deleted or modified, it is always real.

Almost all methodologies provide inclusive rules: things that could possibly be done under all situations. This often results in a far too big list of things that may not or must be done. Agile methods on the other hand try to offer a minimum set of generative rules, which describe things that must be done under all circumstances. Using this small set of principles, appropriate practices for special situations can be generated. Inclusive rules describe someones own interpretation on practices and conditions in all different kind of situations. This of course is subject to changes in time and therefore is doomed to become incomplete. Generative rules, however, need to be interpreted by each individual adhering to these rules, which requires creativity to find ways to solve the problem. Creativity — and not an extended set of written rules — is the best way to manage complex software development problems in a diversity of situations.

2.5.3 Experience with Agile

Our first experience with a software engineering process was, as describe in Section 2.4.2, with the ESA software engineering standard during a software engineering project at the TU/e. The ESA standard is obviously a heavyweight process. We experienced it as heavy, but at that time we had no knowledge of the existence of *lightweight* or agile methodologies. We found that we were writing a lot of documentation, and that the focus lied too much on the process itself instead of the actual product.

For the component development process, we started off with the ESA software engineering standard. We wanted to add more structure to the development of the components and we chose the ESA standard, because we only had experience with that standard. To lighten the process, we switched to the ESA lite standard [European Space Agency, 1996], instead of the full ESA standard. We decided to distill the ESA lite standard to a customized version for the component development process. The resulting process is described in Chapter 4.

After studying the agile movement, the similarities with our customized process became visible immediately. We originally did not design the process to be agile, but the process does fulfill the values and principles defined by the Agile Manifesto, see Chapter 4.

2.6 The Open Source Software Development Model

Open source software shows that reliable, high quality software may be produced quickly and inexpensively. A lot of reports that have been published tend to be somewhat evangelical, concentrating on the genuine success stories associated with the open source approach [Feller and Fitzgerald, 2000]. Recently, quantitative analyzes and empirical studies are conducted on the subject of open source software engineering. The main question is: “Can the open source software development model be classified as software engineering?” To answer this question, first a short history and explanation of open source will be discussed.

2.6.1 Historical Background

Although the term open source was only recently coined, *free* software has a long history. Perens summarizes the history of open source:

> “When computers first reached universities, they were research tools. Software was freely passed around, and programmers were paid for the act of programming, not for the programs themselves. Only later on, when computers reached the business world, did programmers begin to support themselves by restricting the rights to their software and charging fees for each copy. Free Software as a political idea has been popularized by Richard Stallman since 1984, when he formed the Free Software Foundation [Free Software Foundation, 2004a] and its GNU’s Not Unix (GNU) project [Free Software Foundation, 2004c]. Stallman’s premise is that people should have more freedom and should appreciate their freedom. He designed a set of rights that he felt all users should heave, and codified them in the GNU General Public License (GPL) [Free Software Foundation, 2004b]. Stallman punningly christened his license the copyleft — ‘all rights reversed’ — because it leaves the right to copy in place.” [Perens, 1999]

Where proprietary commercial software vendors saw an industry guarding trade secrets that must be tightly protected, Stallman saw scientific knowledge that must be shared and distributed [DiBona *et al.*, 1999]. The scientific method rests on a process of discovery and justification. For scientific results to be justified, they must be replicable. Replication is not possible unless the source is shared: the hypothesis, the test conditions, and the results. Only by sharing source code, peers are enabled to replicate results. To demonstrate the validity of a program to someone, he or she must be provided with the means to compile and run the program. Hence, source code is fundamental to the furthering of computer science and freely available source code is truly necessary for innovation to continue.

The Free Software Foundation policy is strongly ideological and given the idealism of such organization, it is deemed to be anti-commercial and hostile to any profit motive [Feller and Fitzgerald, 2000]. When Stallman is talking about free software, he is actually talking about free as in *libertas*, e.g. free speech and not free as in *gratis*, e.g. free beer. This radical message (the freedom part, not the beer part) led many software companies to reject free software outright. After all, they are in the business of making money, not adding to the body of knowledge of the free software community [DiBona *et al.*, 1999].

> In the spring of 1997, a group of leaders in the free software community assembled in California. This group included Eric Raymond, Tim O'Reilly, and VA Research president Larry Augustin, among others. Their concern was to find a way to promote the ideas surrounding free software to people who had formerly shunned the concept. They were concerned that the Free Software Foundation's anti-business message was keeping the world at large from really appreciating the power of free software.
>
> At Eric Raymond's insistence, the group agreed that what they lacked in large part was a marketing campaign, a campaign devised to win mind share, and not just market share. Out of this discussion came a new term to describe the software they were promoting: open source. A series of guidelines were crafted to describe software that qualified as open source. ... The Open Source Definition (OSD) allows greater liberties with licensing than the GPL does. In particular, the OSD allows greater promiscuity when mixing proprietary and open source software. [DiBona *et al.*, 1999]

A well known dispute in the open source community was going on late in 1998. In an aim to build an object-oriented desktop interface, Troll Technology's Qt library — a proprietary piece of code — was used in the K Desktop Environment (KDE) project. Before the presence of the OSD, Troll Technology would have had to choose between applying the GPL to Qt and maintaining it proprietary. With the new definition of open source, however, Trolltech was able to draw up the Q Public License (QPL) license [Trolltech AS, 2004] meeting the OSD, but still giving them control over the technology they wanted.

2.6.2 The Open Source Definition

Open source software is characterized by several differences to traditional software development and distribution. The OSD [Perens, 1999], articulated by the Open Source Initiative (OSI)[1], is a

[1] *Open source* is a certification mark owned by the OSI (`http://www.opensource.org`)

bill of rights for the computer user. It defines certain rights that a software license must grant, to be certified as open source:

- The right to make copies of the program, and redistribute those copies.
- The right to have access to the software's source code, a necessary preliminary before you can change it.
- The right to make improvements to the program.

These rights are important to the software contributor because they keep all contributors at the same level relative to each other. Perens [Perens, 1999] argues that the reason for the success of this somewhat communist-sounding strategy, although the failure of communism itself is visible around the world, is that the economics of information are fundamentally different from those of other products. Nowadays — in our westernized world — there is very little cost associated with copying a piece of information like a computer program. In comparison, you can't copy a loaf of bread without a pound of flour.

The OSD is not a license itself; a specific license can comply to the OSD or not. When a piece of software is released under a license that satisfies the OSD, the software may be called open source. The canonical version of the OSD can be found at [Open Source Initiative, 2004]. Examples of open source compliant licenses are the GPL and Library GPL (LGPL) license, the Mozilla Public License (MPL) and the X License with related Berkeley Software Distribution (BSD) and Apache licenses.

2.6.3 Open Source Software Engineering

Can open source be classified as Software Engineering? And if so, is it a heavyweight method, an agile method or a method on its own. In the past couple of years there has been a lively debate about open source as software engineering, with many publications on the topic of open source software engineering [Robbins, 2003, Vixie, 1999, Koch, 2004, Feller and Fitzgerald, 2000, Koch and Schneider, 2000, Reis and de Mattos Fortes, 2002, Browne, 1998]. Vixie [Vixie, 1999] tried to analyze whether the process of developing open source projects can be marked as software engineering. Koch has published an article about the relation between agile and open source [Koch, 2004].

Opponents of open source often use terms like cowboy coding, unplanned and undisciplined hacking or similar terms. Many of these arguments often are faced by agile development. Vixie claims that software engineering is "a wider field than writing programs". The seminal moment when a developer changes from *programmer* into *software engineer* is when he realizes that engineering is a field and that he is able to enter that field. But entering the field will require a fundamentally different mindset, and a lot more work. Vixie identifies the elements in the process of engineering as:

1. Identify the requirements.
2. Design a solution that meets the requirements.
3. Modularize the design; plan the implementation.
4. Build; test; deliver; support.

Engineering is an old field, and these elements not only apply to software engineering specifically. Vixie analyzes open source software development according to these different phases, to see whether open source software development can be qualified as software engineering.

Open source developers tend to start a project for a piece of software they wish they had. The requirements are often nailed down on a Usenet discussion forum or via mailing lists. Nonetheless, there is a clear process of negotiation going on amongst developers. The design often is implicit and evolves in the process of implementation. A written down version of the design is rare, but usually there *is* a system design of the software. A detailed design is not very common, the code tends to be the detailed design in open source software. A lot of good and otherwise reusable code gets hidden this way. Although different API documentation systems are developed, they are still not very common in the open source world. The implementation phase is *the* phase what open source development is all about. The opportunity to write code is the primary motivation for practically all open source software development effort ever expended. The way open source software is tested is unique in its field. While the software is being designed and built, experience of end-users are reported back whenever the software is used. In addition, another advantage enjoyed by open source projects is the peer review of dozens or hundreds of other programmers looking for bugs by reading source code rather than just by executing packaged software executables. The support of open source software is rather chaotic. This can keep some users from being willing (or able) to run unfunded open source programs, but it also creates opportunities for consultants or software distributors to sell support contracts or enhanced or commercial versions. More and more companies successfully take this opportunity.

The conclusion of this analysis is that open source software development *can* be software engineering. An example is the Mozilla project which according to this analysis and the research by Reis and Fortes [Reis and de Mattos Fortes, 2002] can be marked as an example of an open source software engineering project. Generally open source developers often succeed for many years before the difference between *writing a program* and software engineering finally catches up with them.

If open source can be qualified as software engineering, the question remains whether it is heavyweight, agile or a methodology on its own. There is an ongoing discussion on this topic. Recent research [Koch, 2004] has shown that there are similarities between agile and open source development. Additionally, empirical data supports these similarities in the emphasis on highly skilled individuals at the center of a self-organizing development team, the acceptance and embrace of change by using short feedback loops and frequent releases of code, and close integration with customers and users [Koch and Schneider, 2000]. However Cockburn [Abrahamsson *et al.*, 2002] notes that open source software development differs from the agile development model in philosophical, economical and team structural aspects. Philosophically and economically, the open source software development model is based on idealistic beliefs. The open source software development model differs from the agile development model in team structural aspects considering the team co-location and personal contact demanded by agile development. The results of this research indicate that the open source development process cannot be qualified as a heavyweight methodology because of the close integration of customers and users and embrace of change.

The software development tools [Robbins, 2003] used in the process are another key factor in the open source software model. There are several common practices that can be found in many open source projects. The most widely adopted open source software engineering tools are

the result of these practices. The features of these tools are aimed at some key practices of the open source methodology and hence less specifically aimed at the common software engineering practices. The aspect of tools influencing the development process, is one of the important issues that will be discussed in this dissertation. Chapter 3 will describe the importance of tools in more detail. The open source development model is one of a kind in this regard.

It can be concluded that the open source software development model shows similarities with the agile development model, and tools play an important role in the open source development process. Because no empirical data exist whether the open source software development model can be qualified as agile, we will treat it is a separate methodology.

2.6.4 Personal Experiences with Open Source

In the software engineering projects that included the authors as team members, much — and often only — open source software was used. Both authors have gained much experience using open source software engineering tools. Examples of open source tools we use frequently and tools that we have used in the past are numerous, including bash, Concurrent Versions System (CVS), GNU's autoconf and automake, make, LaTeX, MetaPost[2], gcc, to name a few. Zwartjes has been working with Linux starting late 1998, using it as his base (development) platform starting from around 2000. Van Geffen has been an active Linux user from around 2000.

Recently both converted to the Gentoo Linux Distribution and are active members of the Gentoo community, through running the unstable (testing) version of the system and reporting and fixing bugs therein. Zwartjes started an open source project called GTK Development Environment (GDE) in October 2000, that aimed to be an Integrated Development Environment (IDE) framework around commonly used tools. This way he had first hand experience with open source developer participation; many people submitted bug fixes and extra features. Unfortunately, university projects and exams took more and more time and left GDE unmaintained four months after it was born. After the acquisition of a new laptop, Zwartjes also joined the ACPI4Asus team for a while, to support the development of the Advanced Configuration and Power Interface (ACPI) driver for Asus laptops.

Open source offers a huge pool of software that can be used freely and that is a valuable resource when the tools for a project are chosen. In Chapter 5 the selection of software engineering tools will be discussed and some of the frequently used open source software engineering tools will be highlighted. The proposal for a method for the development of software components, in Chapter 4, has some influences from the open source model as well.

2.7 Methodology Selection

Khan summarizes the key differences between heavyweight methodologies and agile methodologies [Khan, 2004], illustrated in Table 2.1. An additional row is included that shows the difference in chain of command. The basic difference between agile and heavyweight methodologies is the *weight*. Proponents of agile believe that you cannot achieve agility with heaviness. Some critics

[2]More precisely, the tetex distribution, including both LaTeX and MetaPost, aims to be the TeX system that consists of only free software.

	Agile methodology	**Heavyweight methodology**
Approach	Adaptive	Predictive
Success measurement	Business value	Conformation to plan
Project Size	Small	Large
Management style	Decentralized	Autocratic
Perspective to change	Change adaptability	Change sustainability
Culture	Leadership-collaboration	Command-control
Documentation	Low	Heavy
Emphasis	People-oriented	Process-oriented
Cycles	Numerous	Limited
Domain	Unpredictable and exploratory	Predictable
Team size	Small and creative	Large
Upfront planning	Minimal	Comprehensive
Return on investment	Early in the project	End of the project
Chain of command	Minimal	Considerable

Table 2.1: Differences between agile and heavyweight methodologies

	Agile method	**Heavyweight method**
Objective	Rapid value	High assurance
Scope (requirements)	Subject to change, uncertain, largely emergent unknown	Well known, largely stable
Resources (money, infrastructure)	Uncertain budget, money tight	Sufficient budget
Time	Unclear and not well defined milestones	Clear and defined milestones
Risks	Unknown risks, major impact new technology	Well understood risks, minor impact
Architecture	Design for current needs	Design for current and future needs
Developers	Agile, co-located, collaborative	Process-oriented, adequately skillful
Customers	Collaborative, dedicated, co-located, knowledgeable	Knowledgeable, representative, collaborative
Refactoring, Cost of change	Inexpensive	Expensive

Table 2.2: When to use an agile or heavyweight methodology

refer to agile as a hacking approach. They claim that agility is another fancy and ad-hoc name for lack of planning. However, there is some planning — though limited — involved. The reason for the limit is that it is hard to predict the future, and if the future is going to change, why plan too much? Planning less can be a good approach in extreme situations, but in a predictable environment, it might not deliver the desired result.

Some of the limitations of agile approaches are team and project size. It is hard to create large agile teams, since these methods heavily rely on collaboration and creativity among team members. Furthermore, agile methods are aimed at early return on investment. The standard for success in an agile methodology is early and continuous deliverance of working features and software. In contrast, heavyweight methodologies depend for a great amount on documentation, upfront planning, and close conformance to plan.

Another difference is the way software is developed. In an agile method, software grows with each iteration. Each iteration adds extra value to the final product that is to be delivered. In contrary, heavyweight methods try to formulate all requirements in advance and then build and test the application against the specifications.

Table 2.2 enumerates indications to choose for an agile or heavyweight method. This table was also first published by Khan in [Khan, 2004]. It is good to realize that the indicators in the

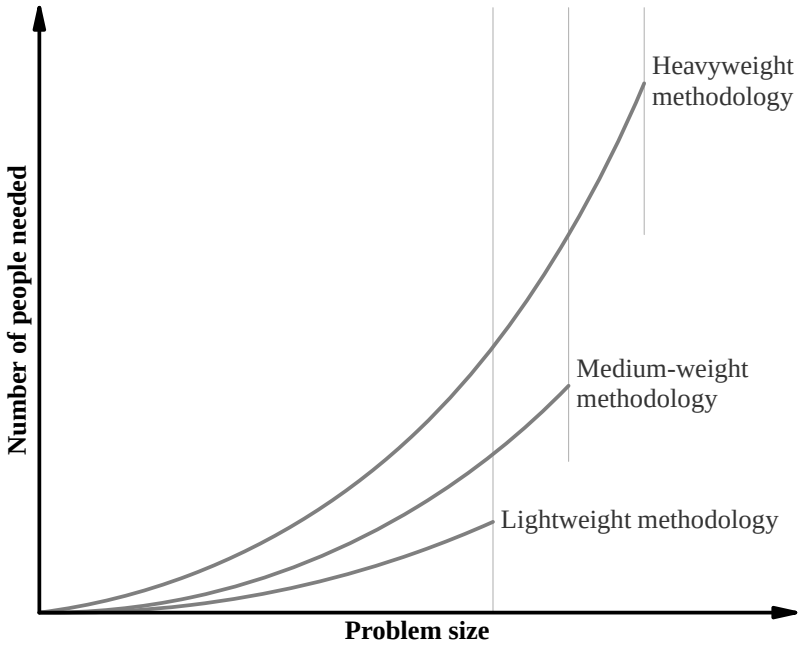


Figure 2.1: Correlation between problem size, number of people and methodology. The vertical lines denote the maximum problem size for the specific methodology.

table are all values at the far end of a spectrum. In practice, you will probably have to deal with values that are somewhere in between of those extrema. In most cases, it will not be possible to say whether you need an agile or heavyweight method. All factors, however, will contribute to your final answer on *how agile or heavyweight* your methodology is or has to be.

Alistair Cockburn has described four principles for methodology selection in "Selecting a Project's Methodology" [Cockburn, 2000]:

- **Principle 1** – A larger group needs a larger methodology.
- **Principle 2** – A more critical system — one whose undetected defects will produce more damage — needs more publicly visible correctness (greater density) in its construction.
- **Principle 3** – A relatively small increase in methodology 'size' or density adds a relatively large amount to the project cost.
- **Principle 4** – The most effective form of communication (for transmitting ideas) is interactive and face-to-face, as at a whiteboard.

One of the reasons to use a methodology is to coordinate people. This can be used to explain the first principle. The larger the group, the larger (or more heavyweight) the methodology. A methodology for a small team should not be expected to work properly for a big team and vice versa. More people requires more weight. Figure 2.1 illustrates this relation between problem size, number of people and methodology, originally drawn by Cockburn [Cockburn, 2000]. Note that there can be a limit for a methodology on the size of a problem that can be solved within reasonable time. A problem with this principle is the fact that no reliable way exists to determine

the problem size at the start of a project or the number of people that are needed to solve it. The number of people can vary for the same project depending on *which* people are in the project team.

The second principle explains the relation between the criticality of the software and the weight of the methodology. Cockburn names 4 levels of criticality in terms of *loss zones* indicating what gets lost when the systems fails: (1) loss of comfort, (2) loss of discretionary money, (3) loss of irreplaceable money, and (4) loss of life. Every next level is more critical than its predecessor. Obviously, a more fine-grained distribution of levels is possible. The criticality is an indication for the *amount* of weight in a methodology.

The third principle states that increasing the methodology with a relatively small amount in size, adds a relatively large amount to the costs of the project. Note that this principle does *not* judge the fact whether or not such an addition to the methodology is *useful*, it only points out the cost of adding elements and control to the methodology. "A pause in the development process to coordinate with other people does not only cost time but also concentration. Updating requirements, design, and test documents is also time consuming." With less weight, people can work with more productivity. The current economic situation and its impact on the available budget for software development projects, caused the cost effectiveness to become one of the most important factors of a project. In business, this is mainly the reason why agile may be preferred.

The last principle is about the communication between people. Communication between people is most effective when it is frequent, face-to-face and interactive [Cockburn, 2000]. If productivity and cost of the project are important issues, then a methodology should emphasize small groups and close contact within a group and between groups. "People that are sitting near each other, with frequent, easy contact, will develop software more easily; that is, the software will be less expensive to develop."

In addition Cockburn addresses two extra factors that may influence the choice for whatever methodology is appropriate: (1) the project priorities, and (2) the methodology designer's peculiarities. "It matters greatly whether the project sponsors or customers want to have the software soon, want it defect free, or want to have the process visible. Each prioritizing produces a different recommendation." Understanding of these priorities is rarely easy. The second factor can be explained by means of the phrase: "All methodology is based on fears." Although it has initially been said by Kent Beck in a dismissive way, it has nevertheless turned out to be relevant. Each element in the process or methodology can be considered a preventive measure against a bad experience some project has had. "Afraid that programmers make mistakes? Hold code reviews. Afraid that designers will leave in the middle of a project? Have them write extensive documentation as they proceed." So if the designer of some methodology would or could express its fears and wishes, much of the design of the methodology would become immediately apparent.

2.8 Process and People

Another important — if not the most important — issue with a software engineering process is the persons that have to adhere to it. Cockburn and Highsmith address several factors in their article on the influence of the human factor in an agile software development environment [Highsmith and Cockburn, 2001b]. "If the people in a project or development team are good enough, they can use almost any process and accomplish their assignment. If they are not good enough, no process will repair their inadequacy." However, a development team does not always consist of the best

developers available. A process will not repair inadequacies, but should guide and bring out the best in people. They continue: "An agile process requires responsive people and organizations. Agile development focuses on the talents and skills of individuals and molds process to specific people and teams, not the other way around." Michael Jackson magnificently points out a truth about the human factor in software development, entitled *Brilliance*:

> Some years ago I spent a week giving an in-house program design course at a manufacturing company in the mid-west of the United States. On the Friday afternoon it was all over. The DP Manager, who had arranged the course and was paying for it out of his budget, asked me into his office.
>
> "What do you think?" he asked. He was asking me to tell him my impressions of his operation and his staff. "Pretty good," I said. "You've got some good people there." Program design courses are hard work; I was very tired; and staff evaluation consultancy is charged extra. Anyway, I knew he really wanted to tell me his own thoughts.
>
> "What did you think of Fred?" he asked. "We all think Fred's brilliant." "He's very clever," I said. "He's not very enthusiastic about methods, but he knows a lot about programming." "Yes," said the DP Manager. He swiveled round in his chair to face a huge flowchart stuck to the wall: about five large sheets of line printer paper, maybe two hundred symbols, hundreds of connecting lines. "Fred did that. It's the build-up of gross pay for our weekly payroll. No one else except Fred understands it." His voice dropped to a reverent hush. "Fred tells me that he's not sure he understands it himself."
>
> "Terrific," I mumbled respectfully. I got the picture clearly. Fred as Frankenstein, Fred the brilliant creator of the uncontrollable monster flowchart. That matched my own impression of Fred very well. "But what about Jane?" I said. "I thought Jane was very good. She picked up the program design ideas very fast."
>
> "Yes," said the DP Manager. "Jane came to us with a great reputation. We thought she was going to be as brilliant as Fred. But she hasn't really proved herself yet. We've given her a few problems that we thought were going to be really tough, but when she finished it turned out they weren't really difficult at all. Most of them turned out pretty simple. She hasn't really proved herself yet — if you see what I mean?"
>
> I saw what he meant.[Jackson, 1995]

2.9 Conclusion

Software engineering is a field that exists for some decades now, and has emerged into a widely used and debated exercise in software development. Three main trends have evolved: heavyweight, agile, and open source software engineering. All of which have some similarities, but also have their own peculiarities. Which one should be used for a project? The heavyweight proponents will claim that you should always use some process and beware of unplanned hacking or cowboy coding. The agile movement will try to convince you to stop the tedious documentation

and focus on the software. The open source developers will want you to give them the code so they can join and help developing.

The choice of methodology depends heavily on the project itself, the team members, and management. In business, cost minimization is often the most important factor. Heavyweight processes are designed to standardize people to the organization, but agile processes are designed to capitalize on each individual and each team's unique strengths [Highsmith and Cockburn, 2001b]. The rationale of choosing a method also lies in the nature of the project. For example, if a malfunction of the software that is to be developed can result in loss of life, more visible correctness is desired. On the other hand, the development of a freely accessible internet information system does not require substantial visible correctness. Without the resources for a project team, it might be suitable to search for volunteers on the internet, and start an open source project. In the previous sections, we have elaborated the properties of the three movements which can be used as a base for the selection of a method.

In the end, people are one of the most important factors within a development environment. It is the people that have to do the final job, independent of what kind of methodology is applied. Roles of people can however differ in different processes. Different methodologies may differ in the way how they utilize the skills of the people that are involved. Considering this human factor in a software engineering process, the choice of process can be vital for a project.

Chapter 3

The Importance of Tools in a Development Process

A really powerful tool changes its user – Donald E. Knuth

The previous chapter introduced software engineering methodology and processes, and illustrated the relation between process and tools in the open source development model. This relation is not only present in open source; tools are important in every software engineering project and thus in every software engineering process. This chapter discusses what software engineering tools are, the historical background of tools, and introduces why, how, and when to efficiently use software engineering tools. Especially the relation between the development process and tools is important in this regard. The selection of a satisfactory set of tools — another important issue concerning software development tools — is described as well. Commonly used tools are categorized and finally our experience with software engineering tools is included in this chapter.

3.1 Introduction

Tools are as old as craftsmanship. In the *Pragmatic Programmer*, Hunt and Thomas point out the analogy with craftsmen: "Every craftsman starts his or her journey with a basic set of good quality tools. A woodworker might need rules, gauges, a couple of saws, some good planes, fine chisels, drills and braces, mallets and clamps. These tools will be lovingly chosen, will be built to last, will perform specific jobs with little overlap with other tools, and, perhaps most important, will feel right in the budding woodworker's hands" [Hunt and Thomas, 1999].

A software tool is a computer program that software developers use to help create or maintain other programs. The term most commonly refers to relatively simple programs that can be used together to accomplish a task. In other words, it is a piece of software that can be used to develop, test, analyze, or maintain a computer program or its documentation. Software development tools

can thus enhance the development process by assisting developers through the complexities of modern software development and by automating tedious tasks. More large scale examples include support for automated building, logging, configuration management, deployment, monitoring, bug tracking and team collaboration — all aspects of an enterprise-scale development effort that are found to be essential by practitioners but are often neglected by the IDEs that are so popular today. Many programmers nowadays only adopt a single power tool, mostly a particular IDE.

Besides these powertools, a lot of smaller tools exist, that are designed for a more specific need. These tools are aimed at doing only a specific task and are designed to be very good at it. By using only an IDE, a developer is missing out on the full capabilities of the environment, for example, version management, and automatic deployment. Tools that support a developer with such tasks are available. Fortunately, some modern IDEs have built-in functionality for third party tools or functionality to support external third party tools. However, these features are still rare and in an early stage of development and a tool often has to support a specific interface to be incorporated into an IDE.

Tool choice today can be bewildering. Software development tools are abundant. Besides the large number of tool vendors, the open source community [Moody, 2002, Raymond, 2001, Torvalds and Diamond, 2002] has been prolific in this regard as well [Robbins, 2003]. However, the available tools are varied in their approaches, their support for critical aspects of different development methodologies, and their ability to work together as a cohesive set. Because open source software is freely available and the number of open source development tools is large and still increasing, open source software is a valuable resource to choose from. In addition, there is an increasing number of industry examples that choose open source tools in favor of proprietary tools.

In this process of open source software adoption in industry, software development tools often lead the way. Finding the open source tools to be generally focused and useful in specific areas, developers use them in a *development tool belt*, or to complement the traditional tool sets purchased from vendors. The open source approach of course holds both promise and risks [Torvalds and Diamond, 2002].

Besides specific open source projects, many software businesses concentrate solely on the development of tools as well. In contrast with open source tools, proprietary tools are often not small applications that focus on a specific task, but a workbench including several tools more or less working together as a bigger, more general tool. Proprietary applications, often are easier to install and learn, provide a user with more support and have a comfortable Graphical User Interface (GUI). But, as they are proprietary, a price must be paid for these extras. Costly tools, however, do not produce better designs. Tools should be judged on their merits, not based on vendor hype, industry dogma, or the aura of the price tag [Hunt and Thomas, 1999].

3.2 Tools and Process

As early as 1975, Brooks writes about sharp tools in his classic, the *Mythical Man Month* [Brooks Jr., 1995]. In his surgical team Brooks reserves one of the members as a toolsmith. Since the existence of programming, tools exist in supporting the act of programming. Likewise since the existence of software engineering, tools emerged to support the process of software engineering; an ongoing challenge facing this profession is the need for average practitioners to adopt powerful

software engineering tools. Starting with the emergence of software engineering as a field of research, increasingly advanced tools have been developed to address the difficulties of software development. Often, these tools addressed the accidental difficulties of development, but some have been aimed at essential difficulties such as management of complexity, communication, visibility and changeability [Brooks Jr., 1995]. The term Computer Aided Software Engineering (CASE) was introduced in the '70s for the tools to speed up the software system building process: a new generation of tools that apply heavyweight engineering principles to the development and analysis of software specifications.

The SEI defines CASE as "the use of computer-based support in the software development process." This definition includes all kinds of computer-based support for any of the managerial, administrative, or technical aspects of any part of a software project. According to the SEI, the history of CASE tools lies in a broader notion of software development. Software development came to be viewed as a large-scale activity, which resulted in a wide range of support tools being developed. The first generation CASE tools concentrated on the automation of isolated tasks. To more effectively support the development process, the individual tasks of the isolated tools are better to be integrated into a *CASE tool environment*. The SEI defines a CASE environment as "a collection of CASE tools and other components together with an integration approach that supports most or all of the interactions that occur among the environment components, and between the users of the environment and the environment itself." The critical part is the facilitation of interaction of the tools: the glue between tools.

Other authors have attempted to make finer grained distinctions between different classes of CASE tools. An example is the distinction between those tools that are interactive by nature (such as a design method support tool) and those that are not (such as a compiler). The former class are sometimes identified as CASE tools. Tools from the latter class are called development tools. Unfortunately, these distinctions are often problematic. In the example, it is difficult to give a simple and consistent definition of *interactive* that is meaningful. Therefore, in this text no distinction between classes of tools is made. In this document, the term *tool* is reserved for a computer program for software developers to help create or maintain other programs and hence includes CASE tools as well. A *tool environment* is a set of several tools glued together.

The roadmap for software engineering tools and environments by Harrison, Ossher and Tarr states that modern software engineering cannot be accomplished without reasonable tool support [Ossher *et al.*, 2000]. The roadmap continues: "Competition is keen throughout the computer industry, and time to market often determines success. There is, therefore, mounting pressure to produce software quickly and at reasonable cost. This usually involves some mix of writing new software and finding, adapting, and integrating existing software. Tool and environment support can have a dramatic effect on how quickly this can be done, on how much it will cost, and on the quality of the result." More important, "[the tools] often determine whether it can be done at all, within realistic economic and other constraints, such as safety and reliability."

The history of a tool environment starts with a small collection of stand-alone tools that can be used in a loosely coordinated fashion to help accomplish software engineering goals. An example of such environment is Unix [Kernighan and Mashey, 1979] and more recently Linux [Goerzen, 2000]. The tools in Unix all accept and output data in standardized formats. As a result, it is possible to interconnect their inputs and outputs loosely, at the developer's discretion. These environments were and still are quite useful, but no real means of integrating tools, coordinating

their executions, or automating common tasks is provided.

In the area of Programming Support Environments (PSEs) the first attempts were made to produce tightly integrated development environments. Because PSEs comprised tightly integrated collections of tools, they were able to overcome many of the problems associated with the earlier, loosely integrated environments. PSEs continue to be extremely useful tools (more recently known as *IDE*), and are still used by the majority of developers today. However, their major limitation — as their name suggests — is that they usually only support *coding* activities. All other major activities and their artifacts, such as requirements engineering, specification, design, testing, and analysis are excluded by most PSEs. The need for integrated support for software engineering activities throughout the software lifecycle resulted in the emergence of *Software Engineering Environments (SEEs)*. They are to the full software engineering lifecycle, what PSEs were to the implementation part. One of the facts that hampered progress on the SEEs is that rapid development dominated the tool scene.

A major line in SEE research was initiated by Osterweil's paper "Software processes are software too" [Osterweil, 1987]. Osterweil's hypothesis — the software engineering process should be treated itself as a piece of software — gave rise to *Process-centered Software Engineering Environments (PSEEs)*. SEEs focus on supporting a specific process model, but PSEEs give up the notion of a predefined process model which has to be applied in every project [Engels *et al.*, 2001]. The price of a more flexible and adaptable PSEE is weaker process support. A compromise is to let a PSEE support a range of process models. Whereas the early PSEEs focused on individual and co-located developer support, recently more attention is given to supporting collaborative software engineering. Examples are GENESIS and OPHELIA [Boldyreff *et al.*, 2003], SourceForge [OSDN, 2004b] and Tigris [Collabnet, Inc., 2004]. The collaborative software engineering environments are prominently present in the open source community.

Robbins analyzes the relation of tools and process in the development of open source software [Robbins, 2003]. He concludes that the open source culture and methodology are conveyed to new developers via the tool set itself, and through the demonstrated usage of these tools in existing projects. A study of CASE tool adoption by Iivari found that adoption correlates negatively with end-user choice, and concludes that successful introduction of CASE tools must be a top-down decision from upper management [Iivari, 1996]. The results of this approach has repeatedly been *shelfware*: software tools that are purchased but not used [Robbins, 2003]. Another research by Stobart and associates shows similar results [Stobart *et al.*, 1993]. Compared to open source tools, these tools were often difficult to use, expensive, and special purpose. The rapid and wide adoption of open source tools stands in stark contrast to the difficulties encountered in adopting the traditional CASE tools [Robbins, 2003]. Robbins references the Fertile Crescent between the Tigris and Euphrate rivers: an agrarian civilization would and did arise first in the location best suited for it. "In other words, the environment helps define the society, and more specifically, the tools help define the method" [Robbins, 2003].

Clearly, the Agile movement [Beck *et al.*, 2001] is an example of the trend predicted by Harrison, Ossher and Tarr [Ossher *et al.*, 2000]. Likewise, the open source movement fits into that trend of an extremely fast-moving and volatile field. Unfortunately, the open source development model is not classified as software engineering by the majority of the academic world (yet). Nevertheless, more and more research is conducted to large open source projects and their results are promising. Examples are the overview of the Mozilla project by Reis and Fortes [Reis and de Mattos Fortes,

2002], and research by Koch and Schneider [Koch and Schneider, 2000]. Beyond doubt the open source development model is an example of modern software engineering.

Tools and process are closely related to each other. The importance of tools in the development process is clear: the tools can either enhance or degenerate the development process. The tool environments from open source projects show that a small collection of stand-alone tools, used in a loosely coordinated fashion, can together form a PSEE. The paradigms from the early software engineering environments applied to recent PSEE research is thereby proved to be successful. As a result, complementing a modern PSE — or IDE — with additional software engineering tools to implement missing functionality yields a PSEE. The previously introduced term *tool environment* actually is the set or a subset of tools that form a PSEE.

3.3 Tool Selection

In their craftsman analogy, Hunt and Thomas continue that a craftsman must choose his or her tools lovingly. Well-chosen and appropriately used tools can contribute greatly to the success of a project. However, inappropriately chosen or ill-used tools are often serious obstacles that work against the development effort. The selection process very often takes place quietly, below the radar, without due consideration as to why that particular combination of products was chosen or how well they fit the overall development task. This leads to decisions that are at best tactical and at worst repeated mistakes.

The selection of open source tools adds its own unique difficulties. Because open source tool builders are often their own (expert) users, installing and maintaining their products takes expertise, and learning how to use them effectively through often sketchy documentation takes patience and time. Moody and Raymond discuss the open source development method and the differences with the proprietary way of developing software in [Moody, 2002, Raymond, 2001]. The necessary knowledge to install, use, and support the chosen tools is too often only in the heads of developers, along with related reasons for choosing one tool over another. This knowledge needs to be captured and made explicit to make tool decisions rational and transparent.

Through their careers, developers go to considerable effort to find, evaluate, choose, and then continue to use a set of tools that supports (and molds) their specific style of development. This often results in different members of a development team using different tools. Although freedom is important in an empowered development team, left uncurbed this leads to a proliferation of tactical solutions, brittle in their support and enigmatic to new team members, maintainers and managers. In order to focus the necessary debates and decisions towards a unified set of tools, candidates need to be well characterized and the criteria for choosing them need to be clear.

In this regard, the document will present an approach that may be followed in managing this process: an abstract model will be proposed that promotes careful consideration and transparent documenting of the rationale behind selecting a set of software development tools. The abstract model is proposed in Chapter 5, including examples, where the model is applied.

3.4 The Basic Tools

A developer has his or her own preferences for tools in the development process — a developer may use a specific tool for a variety of reasons. He or she could be a skilled user of a tool or it can be the standard tool to use within the organization. Maybe the developer does not know or is unwilling to search for alternatives for some kind of tool. A lot of factors influence the choice for a tool.

Whatever the reasons to use — or not use — a specific tool, a developer will always have basic tools for use in the development process. Each tool is the developer's preferred tool of a specific type, or category. Every one of the tools is aimed at a specific purpose and usually is picked out of different implementations to serve the same purpose. Hence, tools can be classified in *tool categories*. The following list summarizes commonly used tools, categorized, and listed in chronological order of usage in the development process.

- **Shells** – Interface various operating system functions and services via a command-line interface. The shell is so called, as it is an outer layer of interface between the user and the innards of the operating system.
- **IDE** – Provide a suite of integrated tools or a framework for integrating tools. An IDE usually contains at least an editor and an integrated compiler.
- **Editors** – Can be used to edit plain text files, for example code source files. This can be a simple line editor up to an editor with support for syntax highlighting, recording macro's etc.
- **Documenting systems** – Can be anything from an advanced WYSIWYG editor to a professional document preparation system. The purpose of a documenting system is to produce digital or paper documents.
- **Version systems** – Help to keep track of different versions of documentation and/or source code. It helps maintaining a code base among multiple developers.
- **Modeling tools** – Provide mechanisms to easily draw diagrams to be used in documentation or as a design tool for the project. Some modeling tools can even generate source code from a diagram.
- **Compilers** – Compile source code into binary form. Except from being integrated into an IDE, a compiler will usually have a command line equivalent. The power of such command line compiler should not be underestimated.
- **Code generators** – Generate code in some language given some input. Very common examples are scanner and parser generators that take a description of a language and generate code in some programming language to scan and parse source files in that language.
- **Code documenting systems** – Provide a clean and easy way to document source code using a special kind of comment in the source code itself. This is also helpful when, for example, a design by contract development method is used. A related, but more powerful, tool category is literate programming.
- **Build systems** – Automate the build process of a project and assist in the process of getting from source code to a public accessible binary form.

- **Testing tools** – Assist in testing the source code of a project. Testing tools can be anything from a simple test bed to an automated testing suite that runs overnight and emails status reports about the software every morning.
- **Debugging** – Provides a methodical process for finding and reducing the number of bugs in a computer program or a piece of electronic hardware to make it work better, or at all. A debugger is a software tool which enables the programmer to monitor the execution of a program, stop it, re-start it, run it in slow motion, change values in memory and even, in some cases, go back in time.
- **Profiling** – Producing detailed information about program execution, such as details about areas of code where most of the execution time is spent, memory usage, code coverage and trace routes.
- **Install systems** – Compile binaries into the form of a package that can be deployed on the Internet or distributed on some media to the public. Provide the user of the application with an easy interface to setup the application on his own computer.
- **Bug- and issue tracking systems** – Give users and developers a consistent interface to report bugs and issues with the software. Acts like an electronic whiteboard.

This is not a complete listing of tool categories, merely a guide to indicate the kind of tools available.

Three interesting studies of tools in a software project are Reis and Fortes' analysis of the open source Mozilla project [Reis and de Mattos Fortes, 2002], Robbins' open source software engineering tool adoption, and Hunt and Thomas' discussion of the basic tools in the toolbox of a developer [Hunt and Thomas, 1999]. We will use our own experiences and examine these studies to find the similarities and differences.

"Every woodworker needs a good, solid, reliable workbench, somewhere to hold work pieces at a convenient height while he or she works them. The workbench becomes the center of the wood shop, the craftsman returning to it time and time again as a piece takes shape" [Hunt and Thomas, 1999]. A programmer or software engineer also needs a decent workbench, which is the command shell. From the shell, the full repertoire of tools can be invoked and all applications, compilers, editors and utilities can be launched. Programmers who have always worked with GUI and IDE might be convinced that everything can be done equally well by pointing and clicking. This is false believe; if you use a GUI or IDE, you won't be able to benefit from all capabilities of your environment. For example it won't be possible to combine tools to create your own customized macros. "The benefit of a GUI is WYSIWYG — what you see is what you get. The disadvantage is WYSIAYG — what you see is *all* you get!" [Hunt and Thomas, 1999].

"Many new programmers make the mistake of adopting a single power tool, such as a particular IDE, and never leave its cozy interface. This really is a mistake. We need to be comfortable beyond the limits imposed by an IDE. The only way to do this is to keep the basic tool set sharp and ready to use." You can become more productive when you use good tools and when you use them in a good way. Hunt and Thomas classify the following tools as basic:

- Version control system
- Bugtracker
- (Advanced) editor

- Debugger
- Code generator
- Text manipulation tool

The Mozilla project [Mozilla, 2004] is dedicated to the development of the Mozilla web browser suite. It has arguably one of the largest communities working on an open source software project today. In the Mozilla Project the following development tools are used:

- Version control system
- Bug and issue tracking tools
- Tool to visualize the code base
- Communication tool

Robbins addresses the commonly used tools within open source software projects [Robbins, 2003]. He focuses on tools that have impact on collaborative development, and omits tools like editors, compilers and debuggers. Robbins names the following categories:

- Version control system
- Issue tracking and technical support
- Technical discussion
- Build system
- Design and code generation tool
- Quality assurance tool
- Collaborative development environment

The three studies have in common a version control system and a bugtracking tool. These tools are nearly always used in a software engineering project, regardless the size of the project and the persons involved.

A version system is merely a different way to store the project's files. The additional time needed to learn the commands to store and retrieve files from a repository are negligible compared to the time to find a convenient way to share code between developers and to be able to request previous versions when you are not using a version control system. Whenever the project team consists of more than one member, a version system will pay off for sure. Even when there's only one developer, a version system provides a very good mechanism to have a backup and history keeping system. Hunt and Thomas describe a source control system as a giant undo button; with a properly configured source code control system, it is possible to always go back to a previous version of the software. Their tip: "Always use source code control".

Immediately after the software is released, a centralized mechanism to store and keep track of bugs in the system is needed: a bug and issue tracking tool. The contents of a bugtracker must be available for the outside world, for the end-users of the system to report a bug and query the status of a bug. Most popular bugtracking tools of today support this feature — an example of a commonly used open source bug and issue tracker is Bugzilla.

In open source projects good communication tools are needed, because project teams are often not co-located. Both Reis and Fortes, and Robbins results illustrate this. The bug and issue tracking system is often (ab)used as a way of communication. The bugtracker is not only a convenient way of communication between developers, but a good mechanism for communication between developers and users as well. Of course the open source community also depends heavily on the standard internet communication tools such as email, mailinglists, newsgroups and internet relay chat. In co-located project teams, most likely face-to-face is the effective way of communication. Cockburn addresses this issue in selecting a project's methodology [Cockburn, 2000].

A build system can be set up by combining several basic tools like version systems, code generation tools, and text manipulation tools. The extra costs to set up a convenient build system are neglectable to the costs of typing in all the necessary commands every time the software is built. The most simplistic build system simply runs the necessary commands to build the software one by one. More advanced build systems require more time to set up, but, depending on the additional features, can pay off on the long term. In addition, a build system ensures that every time the software is built, it is built in exactly the same way. It becomes more difficult to make mistakes, or forget something. Furthermore, a build system can be extended with a release system and a test system, to automatically test and release the software to a website or directly to customers.

The essential tools are a compiler, an editor and a debugger, which are commonly integrated into an IDE. Besides these essential tools, there are three additional tools that are commonly used to improve the development process. Based on our experience and the previous analysis we have found the following:

1. Version control system
2. Bug and issue tracking tool
3. Build system

These are the tools that were found to be important in the analyzed projects and in our own projects.

3.5 Conclusion

It is inevitable that people will make mistakes in the development of software. These errors often cost a lot of time and resources to fix and are serious obstacles in the process of development. Tools can decrease the development effort by assisting in managing mistakes in three ways:

1. By preventing errors
2. By detecting and locating errors
3. By fixing errors

The *Pragmatic Programmer* points out: "Tools amplify your talent. The better your tools, and the better you know how to use them, the more productive you can be." When there are tasks that can be automated, a tool can prevent mistakes in such tedious job. However, it can cost a lot of effort incorporating a tool in the existing development process or to train developers to use a certain tool. The process of tool selection is therefore very important.

The evolution of software engineering methodology, processes, tools and environments shows that tool selection can better not be an upper management decision. Expensive CASE tools degraded to shelfware is an example of this repeatedly made mistake. Tools and process must be one: the tools help define the method.

A PSEE can be implemented by complementing the missing features of a PSE. In practice, external software development tools can be used to supplement the widespread used IDEs. The open source community provides a growing amount of tools as a resource to choose the supplements. This is a result of the close relation between tools and methodology in the open source development model, which hopefully will set an example for further research.

Chapter 4

An Agile Method for the Component Development Process

Frustra fit per plura quod potest fieri per pauciora (It is vain to do with more what can be done with less) – William of Ockham (circa 1285 – 1349)

This chapter contains the documentation of the process for the development of software components as described in Chapter 1. It describes the guidelines for the process that should be followed by the component developers by outlining the phases that are iterated in the course of a project. The tasks and deliverables of each phase are addressed, as well as the management issues in a phase. In addition, the templates for product documentation are given. The proposed method is aimed to *assist* in managing the course of a project wherever possible. The purpose of this chapter is not only to describe the guidelines to follow, but also to motivate why to follow them.

4.1 Introduction

The process that is described in this chapter, is originally designed for the development of software components. "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties" [Szyperski *et al.*, 2002]. In other words, a software component can be defined as a package or a set of one or more objects including the description of how they work. The set of objects represents a coherent set of features — the objects are the interface of the component — that can be used in an application or another component. A component can implement one or more design patterns for example, described by Gamma and associates [Gamma *et al.*, 1995]. Independent deployment of a component implies that the component can be fixed and enhanced separately and independently from the application, or component, it is used in.

The development of a component is started by gathering information about its task — what the component is supposed to do — by interviewing the developers who will use the component. This information is used to write down the requirements. The next step is to design and implement the component. A component is deployed as soon as possible and the component developers continuously respond to feedback from end-users. The early and frequent deployment is derived from the open source paradigm *release early and release often* [Raymond, 2001]. Requirements are added as necessary, and bugs and new features are identified, fixed, and implemented as soon as possible.

The components are developed by a separate development team, technically operating on its own. This imposes requirements on the availability and quality of the developed components. To minimize costs of the component development process, open source development tools are preferred. In contrary to most proprietary tools, there are no restrictions on the amount of people that can use the tool. Two other advantages in this case are: (1) most of these tools are released often — new features are readily available and bugs are solved swiftly — and (2) bugs and new features can be added at will because the source code is available.

In the following sections, the process and its phases are described. The relevant software engineering methodologies that affected the process were discussed in Chapter 2. For each phase, a section follows to elaborate the tasks and deliverables for that phase.

4.2 Process Evolution

The standard described in this chapter has evolved from the ESA software engineering standard. Our experiences with the ESA software engineering standards, elaborated in Section 2.4.2, heavily influenced the work on the components. The first standard that we applied for development of the components was the full ESA standard.

As prescribed by the ESA standard, a project was started by creating a set of management documents — a set of documents for each new component, because each project involves one component. As a result, these management documents were created by copying the documents from a previous project and changing the project name, among other minor details. The only non-straightforward task, involved creating a detailed plan. Next, the component's product documents were created: a User Requirements Document (URD), Software Requirements Document (SRD), Architectural Design Document (ADD), and Detailed Design Document (DDD). A lot of sections in the product documents could be copied from previous projects as well. Much text was already outdated by the time the component was deployed.

After a couple of projects, this process seemed not to be optimal, in this context. In comparison with, for example, air traffic control systems, and military command and control systems, the software components are quite small. A heavyweight process, like the ESA standard, is designed for the development of the former. Such process has many things built into it because of the need for managing larger projects with subcontractors, many channels of communication, and many — perhaps conflicting — requirements. None of which are present in the development of our software components.

Heavyweight methods adopt more of a codification of what needs to be done, including things like contract management, precise specifications, and formal sign-offs, which are absolutely nec-

essary in larger efforts, but cause unnecessary overhead in the component development context. Especially over-documenting caused much overhead in our case. A document should have only one goal, which is to assist in accomplishing the desired result: deliver a working software product and make it maintainable.

To increase productivity by decreasing documentation overhead, we switched to the ESA lite standard [European Space Agency, 1996]. All three criteria (listed in Section 2.4) to use this simplified version of the ESA standard apply: (1) the development of a component requires less than two man years of effort, (2) the number of team members varies between one and four, and (3) the number of lines of code is less than 10,000. Moreover, the components are not critical in the sense of loss of life or big amounts of money. These arguments validate the use of the ESA lite standard in favor of the full version of the ESA standard.

Unfortunately, the ESA lite standard still prescribes management documents to be written for each project. As mentioned before, apart from a detailed plan, the management tasks and documents are essentially the same for each project. Adding the small size of the development teams, it is not useful to write the management documents for each project. We decided to alter the ESA lite standard, and define a new process based on the ESA lite standard with our own modifications. The first step towards a more agile approach, was to eliminate separate management documentation per project. We started molding the process to the project(s) and not the other way around.

It was decided to alter the ESA lite standard, instead of replacing it with an agile process, because there were already several projects running with the ESA lite standard. In addition, all development team members had become familiar with the ESA lite process. The shift from ESA lite to an agile approach, however, had already partly begun. The implementation techniques from XP and pragmatic programming, such as pair-programming and refactoring, had already been successfully applied in the implementation phase of several projects.

The adaptability, that is omnipresent in agile processes, was the most important aspect that was still lacking in the component development process. For example, it was often necessary to change the requirements or the design, while the source code was constructed for a component, and moreover, after delivery of the software, the customer frequently wants to change the requirements, as well. The following list summarizes the key changes to the process, to add adaptability:

- Management activities are essentially the same for each project, can be incorporated in the process, are minimized, and are documented in the process standard.
- No detailed upfront plan is made, the customer's wishes determine schedule and can be adapted at any given moment.
- Adaption of an iterative approach and shortened release cycles. The possibility to fall back to a previous phase — without unnecessary restrictions and overhead — is added.
- Closer interaction with customer(s) and end-users; in each phase customer(s) and end-users are highly involved.
- Less extensive product documentation, process focuses on actual product, not documentation.
- Single project team is responsible for the entire project, and all tasks are fulfilled by that single team.

These changes reflect the values of the Agile Manifesto [Beck *et al.*, 2001]. Individuals and interactions are valued over processes and tools, with minimized management and a decentralized approach. Working software is valued over comprehensive documentation by minimizing documentation and adapting continuous and frequent releases of the product. Customer collaboration is valued over contract negotiation by a low chain of command (customer and developer interact directly) and by welcoming changes in every phase of a project. With an iterative approach and shortened release cycles, responding to change is valued over following a plan. The last section of this chapter will compare in detail the process with the Agile Manifesto.

Three aspects of the ESA lite standard remain in the new process: (1) a separation into phases, (2) a requirements document, and (3) a design document. The process that is described in this chapter is divided into phases, although not as strictly as in the ESA lite process. The following section will go into more detail of a project's phases. The requirements and design document are described in Sections 4.4 and 4.5 respectively.

4.3 Process Overview

A project iterates through different stages. The process is divided into phases according to these stages. Such a phase indicates what tasks are currently being worked on in the course of a project. This section gives a general overview of these phases and how they are connected. The stages of a project should be iterated consecutively, but it is possible to fall back to an earlier stage after the project progressed into a subsequent stage. Note that no phase should be skipped in forward direction; every phase should be traversed to reach the maintenance phase. Figure 4.1 shows the four phases and the relations between these phases. The four phases are:

- **Exploration Phase (EP)** – The initial phase of a project, a project is started and the user requirements are gathered in this phase.
- **Design Phase (DP)** – The user requirements are analyzed and a component decomposition, with accompanying design models, is drawn up in this phase.
- **Construction Phase (CP)** – The requirements are implemented according to the design made in the previous phase. Test versions of the product are released to the customer as often as possible. At the end of this phase the product must be finished.
- **Maintenance Phase (MP)** – The final product is officially released to the end-users. Bugs are collected and fixed, and the end-users are provided with support.

The phases are described in full detail in the following sections. All deliverables and tasks — including management — are described per phase in a separate sections. The process is meant to be a *guideline* for the component developer, and so are these phases. Deviation from the process and documents is allowed, as long as there is a profound reason for it, and whenever the change contributes positively to the — outcome of the — project. To this extent, the tables from Sections 4.4.3, 4.5.4 and 4.5.5, that provide the table of contents for the documents that are to be created in the exploration and design phase, should be regarded as a guideline or mold, that have proved to be suitable for most component projects.

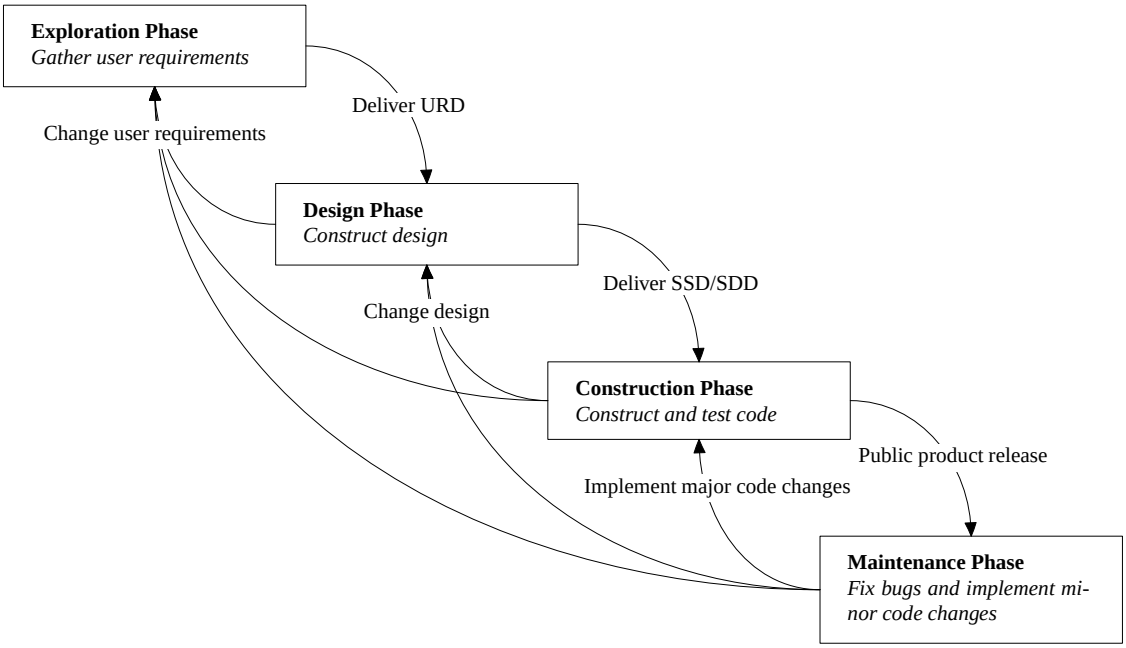


Figure 4.1: This figure depicts the relation between the phases in the process and transitions between them.

4.4 Exploration Phase

In the exploration phase the user requirements are gathered. The URD is the deliverable of this phase and contains the user requirements. Gathering the requirements is the first step of the development process. Issues concerning user requirements are detailed in Section 4.4.1. The overall structure of the URD is written down and clarified in section 4.4.3. If necessary, a prototype of the software can be created in the exploration phase, which will be described in more detail in section 4.4.2.

4.4.1 User Requirements

The main goal of the exploration phase is to define the user requirements. The user requirements form an understanding between the customer and the developers. For the customer, the requirements put down in writing the *minimal* functionality of the software. For the developers, they are the base for the design. The user requirements are written down in the URD, which is described in further detail in Section 4.4.3.

It is difficult to find out exactly what a customer expects from the software that is to be developed. Customers often do not know what they really want. Developers often *think* they know what the customer wants, though the customer has something different in mind. To prevent these situations, it is important that the customer and the developer communicate face-to-face with each other by means of meetings and interviews. It is not realistic to expect a user to define and formulate all its wishes in advance. A customer or developer might think of some new features or

other formulations when (part of) the implementation gets available. The developer should try to get as much information as possible from the customer, guide and support his thinking and try to anticipate on changes that might surface in the future.

Prototyping can assist in the process of formulating the requirements. It is up to the developer to convince the customer that it is important to have a complete set of requirements in advance. The design that will be created in the design phase is based on these requirements. The customer should realize that future changes in the requirements can have a significant effect on the design and consequently the implementation.

Requirements have to be written down such that misunderstandings are avoided. User requirements should be clear and verifiable to make sure that the final software product will comply with the requirements. The user requirements are complete and accurate if the URD reflects the real needs of the customer.

There are two basic categories of user requirements: capability requirements and constraint requirements. "Capability requirements describe the process to be supported by the software. Simply stated: they describe *what* the software is supposed to do according to the customer." [European Space Agency, 1995b] Each capability requirement should specify an operation, or sequence of operations, that the software will be able to perform. Requirements should not be too complex. Requirements that describe too much functionality should be divided in multiple requirements. It is also important to make a clear grouping of the requirements. Requirements that functionally relate to each other must be kept together.

Constraint requirements impose restrictions on the way the user requirements are to be met. These constraints can relate to interfaces of the software, quality, resources and timescales. Interfaces are a very important kind of constraint requirements: an interface defines the shared boundary between two systems of the software and describes what information is exchanged and how it is done. The following list summarizes common categories of constraint requirements, based on the categories as defined in the ESA standard [European Space Agency, 1995b]:

- **Performance** – Define quantitative statements that may describe amplitude, speed, or accuracy. The amplitude states how much is needed of a capability at any moment in time. The speed defines how fast a specific operation is to be performed. The accuracy defines the difference that is allowed between the intended and the actual output of some operation.
- **Communication Interfaces** – Specify the networks and network protocols that have to be used together with performance attributes if necessary.
- **Hardware Interfaces** – Define all, or part of, the computer hardware that the system is supposed to run on.
- **Software Interfaces** – Specify the software the system has to be compatible with, such as other applications, operating systems, programming languages and compilers.
- **Human-Computer Interaction** – Specify elements and aspects of the user interface of the system, such as style, messages and responsiveness.
- **Adaptability** – Define how easy the system can deal with changes in the requirements. Extra design work may be needed to fulfill these requirements.
- **Availability** – Specify when the system in operation must be available, for example between 9 o'clock in the morning and 5 o'clock in the afternoon or an average availability of 98% per year.

- **Portability** – Describe what is required when the system needs to be ported between two environments. An environment can be a different hardware base or operating system for example.
- **Security** – Define the measures that have to be taken against threats to the confidentiality, integrity and availability of the system, such as hackers, virus intrusions, fires, computer breakdowns and illegal activities of unauthorized users.
- **Safety** – Define the needs of users to be protected against potential problems such as hardware and software failures.
- **Standards** – Reference the applicable documents that define the standards of the system. A standard can be a process or product standard. The purpose of this section is not to mention the software engineering standard that is applied during the project, but to mention standards like export file formats and so on.
- **Resources** – Describe the available resources for building the system, such as the required hardware that the software is supposed to run on.

The categories mentioned above reflect the available categories of constraint requirements. Note that only the categories applicable to the system must be included in the constraint requirements section of the URD.

4.4.2 Prototyping

Requirements and models are usually static entities. In some cases, it may be useful to construct a dynamic model that is executable to verify parts of the system. The process of creating a dynamic model is called prototyping.

The prototype reflects the view on the system of the prototype's creator, which can either be the customer, the developer, or both. Prototyping increases mutual understanding of the system, and tries to match the view on the system of the developer and customer. New requirements may surface in the process of creating the prototype, and existing requirements may turn out to be incomplete or even erroneous.

Prototyping can be particularly useful when visual user interface elements need to be described. In this case, the prototype can be a valuable extension to the user requirements or replace some requirements completely.

4.4.3 The User Requirements Document

The URD summarizes what the system that is to be developed is supposed to do. It contains a general description of what the customer expects from the software system and a section with the actual capability and constraint requirements. The URD should be arranged according to the following table of contents.

URD Table of Contents

1.1 Definitions, acronyms, and abbreviations
1.2 References

2 General Description

2.1 General capabilities
2.2 General constraints
2.3 User characteristics
2.4 Operational environment

3 Specific Requirements

3.1 Capability requirements
3.2 Constraint requirements

In addition, a Document Status Sheet (DSS) and table of contents should be included as unnumbered chapters just before the first chapter. The DSS is a table that contains the status of the document. It should include the title, identification, authors, and approval status of the document. The approval status of the document can be:

- **Draft** – The document is not yet internally accepted by all team members.
- **Internally accepted** – The document is accepted by all team members.
- **Approved** – The document is accepted by the customer.

In addition, the DSS contains a revision history. The document revision number is formatted as *x.y.z*. Here, *x* denotes the major release number, which is increased to one when the document is externally approved. Hereafter, the number is increased with one every time the document's content significantly changes. The number denoted by *y* is increased to one when the document is internally accepted. The last number, *z* is increased after every minor change to the document. The document revision history contains each revision number including the date of release and reason for change, starting from revision 1.0.0. An example of a DSS for an URD can be seen in Table 4.1.

URD/1 Introduction

This chapter should provide a general introduction about the project and the document.

URD/1.1 Definitions, acronyms, and abbreviations

This section should provide a complete list of definitions, acronyms, and abbreviation that are used throughout the document. When definitions, acronyms, or abbreviations are listed in other documents, then references to those documents should be listed here as well.

URD/1.2 References

This section should list the relevant references to other documents.

Document Status Sheet		
Title	User Requirements Document: Project name	
Identification	Project identifier	
Author(s)	Author(s)	
Status	Draft / Internally Approved / Approved	
Revision	**Date**	**Reason for Change**
#.#.#	Month ##, Year	Author(s): Revision comment
#.#.#	Month ##, Year	Author(s): Another revision comment
#.#.#	Month ##, Year	Author(s): And yet another revision comment

Table 4.1: An example of a DSS.

URD/2 General Description

General factors that influence the requirements for the system should be provided in this chapter. This chapter does not state specific requirements.

URD/2.1 General capabilities

This section describes in natural language what the software is supposed to do. An overview of the main capabilities and the process that is to be supported by the software are given.

URD/2.2 General constraints

In this section, all developers' limitations for building the software are summarized. This section does not point out specific requirements or constraints on the design. It should, however, explain why certain constraints exist.

URD/2.3 User characteristics

This section describes the general characteristics of the end-users that may influence the requirements of the system that is to be developed. Characteristics of the users that can be mentioned are the educational level, language, experience, and technical expertise. In addition, it could be important to know the amount of users that will use the system.

URD/2.4 Operational environment

The environment, in which the system is to operate, is described in this section. A narrative description can be strengthened by context diagrams in which the relation to external systems could be clarified.

URD/3 Specific Requirements

This section will summarize the specific requirements. Each requirement is uniquely identified and has a priority that indicates how desired that specific requirements is. User requirements should be verifiable and clearly written down using proper language. The requirements are divided in two sections containing the capability requirements and the constraint requirements.

URD/3.1 Capability requirements

This section summarizes all capability requirements for the system. This section can be divided in multiple subsections to structure the requirements and increase readability.

URD/3.2 Constraint requirements

The constraint requirements are listed in this section. For each constraint requirements category, as defined in Section 4.4.1, a subsection should be present if there are requirements in this category.

4.4.4 Review and Management Practices

The URD should be constantly under review by all team members. The status of the document progresses to internally approved, at the moment the team agrees on the content of the document. For external approval, the document should be reviewed by the customer and employees that are not in the project team. However, customer and non team member employees should be continuously involved in the process of creating the document.

Only when the project must be restarted — when the requirements need to be drastically revisited due to new techniques, new insights, inadequate or changed requirements — the document will have a status of draft again. Other changes will only lead to a new revision of the external approved document. When the requirements in the document change, the document should be reviewed by the customer again, before being publicly released.

The URD should be under version management control from the first draft. Every version of the document after external approval should be tagged in the version management system. External approved revision of the document should be publicly released.

4.5 Design Phase

The design phase can be considered as the problem analysis phase in the software development cycle. The user requirements from the exploration phase are examined and a *design solution* is produced accordingly. This design solution consists of a logical model, optionally a set of software requirements, and an architectural design.

The deliverable in this phase is a Software Specification Document (SSD) or a Software Design Document (SDD), which contains a description of the design solution. The SDD is a less extensive version of the SSD, intended for less critical or small projects. For critical or large projects the SSD is recommended. The criticality of projects and methodology selection is described in

Section 2.7, and the applicability of the process is described in Section 4.9. The practical differences between SSD and SDD will be described in more detail in this section.

The first step in the creation of the design solution, is building the logical model. This model can be used to optionally specify a structured set of software requirements, depending on the criticality of the project. For a critical and larger project, it is recommended to formulate the software requirements. For less critical and smaller projects, this process may be too time consuming, and the effort put into formulating the software requirements, does not weigh up to the benefits. The design solution is completed with an architectural design. The following sections will describe the design solution in more detail.

All team members that did not take part in the exploration phase writing formulating the requirements, need to examine the URD carefully before participating in the design phase.

4.5.1 Logical Model

The *logical model* is an implementation independent model reflecting the customer's wishes — what the system must do — organized in a hierarchical fashion. Because the end product will be an object-oriented component or application consisting of several classes, the logical model can best be constructed by applying an object-oriented analysis method. Well known object-oriented analysis methods are Rumbaugh [Rumbaugh *et al.*, 1991], Shlaer-Mellor [Shlaer, 1988, Shlaer and Mellor, 1992], Coad-Yourdon [Coad and Yourdon, 1991] and Booch [Booch, 1991]. The modern and broadly used UML can be used to capture the logical model [Object Management Group, 2004]. The project team is however free in choice of both modeling technique and language.

The logical model should be a functional decomposition of the software. The main goal is to identify the important subcomponents. The word component in subcomponent here does not mean software component, but the normal English definition "part of a larger whole" [Oxford University, 6th edition 2002]. The larger whole — the project — in the definition can be a software component. These subcomponents should consist of one or more classes that will be detailed in the architectural design. The logical model only identifies the subcomponents' classes and their properties.

4.5.2 Software Requirements

The software requirements are the developer's view on the problem, and are described according the logical model. The hierarchy from the logical model must be used to structure the software requirements. There are two types of software requirements: (1) functional requirements, and (2) non-functional requirements. The functional requirements specify the functions that a subcomponent or a subcomponent's class must be able to perform.

Non functional requirements are usually related to a functional requirement and specify a quantitative statement about the functional requirement. The non-functional requirements can be categorized. The following list shows common categories of non-functional requirements:

- **Performance requirements** – Specify a quantitative measure for a defined function — for example speed, accuracy, rate and frequency. A performance requirement may specify a range of tolerated values.

- **Interface requirements** – Specify hardware and software that the system must be able to interact or communicate with, and protocol descriptions and other interfaces that the system must support.
- **Operational requirements** – Describe specific requirements for the interaction of the system with the operational environment.
- **Resource requirements** – Specify special needs for physical resources, such as memory, hard disk space, and CPU power.
- **Security requirements** – Describe the special needs for the security of the software, they specify how the system is secured against possible threats.
- **Portability requirements** – Specify the possibilities of operating the software in other environments.
- **Quality requirements** – Define the quality attributes of the software — the use of specific regulations, standards, or qualification of third party systems for example.
- **Reliability requirements** – Specify the Mean Time Between Failure (MTBF) of the system, if applicable.
- **Maintainability requirements** – Specify the degree of adaptability of the system.
- **Safety requirements** – Specify the requirements to reduce the possibility of damage that can follow from failure.

These categories may give an indication of what non-functional requirements should be included in the software requirements. Obviously only those categories that are applicable to the software should be included.

Both functional and non-functional requirements should be verifiable; there should be some way to check whether the software implements the software requirement or not. If a requirement is not verifiable, it is unusable.

Software requirements should be formulated only for larger and more critical projects. The software requirements will assist in progressing from logical model to architectural design. For small and less critical projects the software requirements mostly provide no, or only little, additional information that will influence the architectural design. In those projects, the team can continue from the logical model to the architectural design.

4.5.3 Architectural Design

The *architectural design* is the actual implementation solution to the problem described by the software requirements and user requirements. The architectural design consists of a *physical model* and description of the interfaces and functionality of the subcomponents and the subcomponents' classes, using implementation terminology. The physical model is constructed according to the logical model, software requirements, or user requirements.

Each subcomponent in the logical model will have a corresponding part of the physical model in the architectural design. The model may be described with UML [Object Management Group, 2004], for example. It should provide a framework for development in the next phase, the construction phase. This framework is provided by the model by describing the public interfaces of the subcomponents. These interface descriptions — classes, attributes, and methods — should be

clarified when necessary using natural language. The adaptability of the design is important, in case the customer needs changes in requirements.

The physical model should cover all software requirements for an SSD, or user requirements in case an SDD is chosen.

4.5.4 The Software Specification Document

The purpose of the Software Specification Document (SSD) is to describe what the product is supposed to do. It should cover all the requirements stated in the URD. It will contain the design solution that is used in the construction phase to implement the software. The first part of the SSD covers the developer's view of the software, without using actual implementation terminology — Chapter 2 and 3. The second part — Chapter 4 — an object-oriented design for the software using implementation details. The SSD should be compiled according to the following table of contents.

SSD Table of Contents

1 Introduction

- 1.1 Definitions, acronyms, and abbreviations
- 1.2 References

2 Model Overview

- 2.1 Decomposition description

3 Specific Requirements

4 Design

- 4.1 Global types and classes
- 4.n Subcomponent
  - 4.n.1 Design model
  - 4.n.2 Function
  - 4.n.3 Dependencies

In addition, a DSS and the table of contents should be inserted just before Chapter 1 as unnumbered chapters. The purpose and use of a DSS is already described in the exploration phase, Section 4.4. The document status sheet contains the status and revision number of the document, using the same revision numbering conventions as for the URD.

SSD/1 Introduction

This chapter should contain all introductory material about the project and the document.

SSD/1.1 Definitions, acronyms, and abbreviations

All definitions, acronyms, and abbreviation used throughout the document should be listed here, or references to documents that contain the appropriate definition or term should be included. Ambiguous terms should always be inserted here, to define a single meaning for the term.

SSD/1.2 References

This section should list the relevant references to other documents.

SSD/2 Model Overview

This chapter contains the visualization of the logical model, including a swift overview of the subcomponents.

SSD/2.1 Decomposition description

The section should contain a more detailed decomposition description of the logical model. The section can be in the form of a walk-through the model level-by-level, highlighting important facts about the subcomponents and optionally describing the subcomponent's classes.

SSD/3 Software Requirements

This chapter contains the software requirements. Each requirement should have an identifier and priority. The chapter may be divided into sections to structure the requirements.

SSD/4 Design

This chapter contains the physical model and clarification. If applicable, a (visual) overview of the physical model can be included here. The subcomponents of the model are elaborated in a separate section per subcomponent.

SSD/4.1 Global types and classes

The global (implementation specific) types and classes (such as exception classes), applicable to all subcomponents, are listed in this section. Custom data types, that are used to define the interfaces of subcomponent, should be included as well.

SSD/4.n Subcomponent

For each subcomponent a section should be inserted, with the identification of the subcomponent as section title. A brief introduction to the subcomponent should be inserted here.

SSD/4.n.1 Design model

This subsection should include a visualization of the part of the physical model representing this subcomponent, including a clarification of the model if necessary.

SSD/4.n.2 Function

This section describes the function of the subcomponent, by stating what the subcomponent does and how it should do it. The process within the component can be outlined, including information that is transmitted and stored. The input and output of the subcomponent may be clarified, as well as a description of the attributes of the subcomponent including the range of possible values and initial value if applicable.

SSD/4.n.3 Dependencies

All dependencies, internal and external, should be listed in this section. Dependencies on other subcomponents, other projects, third party software, hardware, and so on, should all be clarified here.

4.5.5 The Software Design Document

A Software Design Document (SDD) is a less extensive version of the SSD. It does not contain a software requirements section. Obviously, the architectural design must cover all user requirements instead of all software requirements in the SDD.

SDD Table of Contents

1 Introduction

- 1.1 Definitions, acronyms, and abbreviations
- 1.2 References

2 Model Overview

- 2.1 Decomposition description

3 Design

- 3.1 Global types and classes
- 3.n Subcomponent
  - 3.n.1 Design model
  - 3.n.2 Function
  - 3.n.3 Dependencies

In the SDD a DSS and table of contents must be included before the actual content as well.

4.5.6 Review and Management Practices

The SSD or SDD should be constantly under review by all members of the project team. When the team agrees on the content of the document, the document's status progresses to internally approved. For external approval the document should be reviewed by the customer and employees not in the project team. However, customer and non team member employees should be continuously involved in the process of creating the document.

Only when the project must be restarted — when the requirements and design need be drastically revisited due to new techniques, new insights, inadequate or changed requirements — the document will have a status of draft again. Other changes will only lead to a new revision of the external approved document; the document must be reviewed by the team members before being publicly released again. Whenever the design leads to changes in the user requirements, it is possible to reiterate the exploration phase, with the restriction that the customer agrees with the changes in the user requirements.

The SSD or SDD should be under version management control from the very first draft. Every version of the document, after external approval, must be tagged in the version management system. External approved revision of the document must be deployed.

4.6 Construction Phase

The construction phase is the phase, in which the product is implemented. The developers write the source code, and document and test the software, following the design from the design phase. The programming tasks can be divided amongst the developers on basis of the subcomponents specified in the SSD or SDD. The outputs of this phase are source code, API documentation and a Software User Manual (SUM), if applicable.

4.6.1 Implementation and Testing

The developers that were not part of the team during the exploration or design phase should read the URD and SSD or SDD. The implementation of the final product, producing the source code, is the most important job in this phase. To increase source code readability, a set of coding standards should be prepared. Including coding standards, several other techniques can help in the process and a number of important techniques from Extreme Programming and Pragmatic Programming are included next.

The *pair programming* technique applied in extreme programming [Auer and Miller, 2003] can be applied in this process as well. This technique requires two software engineers to participate in a combined development effort at one workstation. Benefits of pair programming are:

- Increased discipline. Pairing partners are more likely to "do the right thing" and are less likely to take long breaks.
- Better code. Pairing partners tend to come up with higher quality designs.
- Resilient flow. Pairing leads to a different kind of flow than programming alone, but it does lead to flow. Pairing flow happens more quickly: one programmer asks the other, "What

were we working on?" Pairing flow is also more resilient to interruptions: one programmer deals with the interruption while the other keeps working.

- Improved morale. Pair programming, done well, is much more enjoyable than programming alone, done well.
- Collective code ownership. When everyone on a project is pair programming, and pairs rotate frequently, everybody gains a working knowledge of the entire code base.
- Mentoring. Everyone, even junior programmers, has knowledge that others do not. Pair programming is a painless way of spreading that knowledge.
- Team cohesion. People get to know each other more quickly when pair programming.
- Fewer interruptions. People are more reluctant to interrupt a pair than they are to interrupt someone working alone.

Studies have shown that after training for the people skills involved, two programmers are more than twice as productive as one for a given task. More information about pair programming can be found in the XP series [Beck, 1999, Auer and Miller, 2003, Jeffries *et al.*, 2000].

The *Pragmatic Programmer* actually is a compendium of useful programming tips and tricks [Hunt and Thomas, 1999]. The *Don't Repeat Yourself (DRY) Principle*, *Broken Window Theory* and *Refactoring* are three valuable highlights from Hunt and Thomas' book. Refactoring, a technique also applied in XP, tells a developer to not hesitate to change the workings of a piece of code. New insights, new performance requirements or a nonorthogonal design are all reasons to refactor. Refactoring should be done with care, taking slow steps and without adding features concurrently.

The broken window theory tells something about software rot. The theory originates from research in the field of crime and urban decay. A broken window, left unrepaired for a substantial length of time, can cause a sense of abandonment. Another window gets broken, people start littering, graffiti appears and serious structural damage begins. Then the building becomes damaged beyond the owner's desire to fix it, and abandonment becomes reality. The same holds for broken windows in software — a bad design, a poor piece of code, or terrible error handling. Broken windows should be fixed as soon as they are discovered. System deterioration and software rot should be avoided at all costs.

The DRY principle states: "Every piece of knowledge must have a single, unambiguous, authoritative representation within a system" [Hunt and Thomas, 1999]. Duplication can easily lead to mistakes: "The alternative is to have the same thing expressed in two or more places. If you change one, you have to remember to change the others, or, like the alien computers, your program will be brought to its knees by a contradiction. It isn't a question whether you'll remember: it's a question when you'll forget" [Hunt and Thomas, 1999].

Developers are encouraged to write unit tests for the subcomponents specified in the SSD or SDD. Unit testing is described in both the XP method [Auer and Miller, 2003] and the *Pragmatic Programmer* [Hunt and Thomas, 1999]. The final product — the component — can be viewed as a unit as well, because it is a part in an application or another component. Therefore a test should be written that tests the workings of the system, in case the developed product is a component.

4.6.2 API Documentation and User Manual

In case the project comprises a component, API documentation should be written. In other cases besides API documentation, a software user manual may be written. API documentation should be in the source code itself according to the defined coding standard, describing how to place API comment. If possible, a tool should be used that is able to generate online documentation and paper documentation of the source code.

The API documentation describes the (inner) workings of the source code and how to use the software as a developer. Each class including its properties and methods should be described. For methods, its parameters and return value should be described as well. The descriptions in the API documents should be clear enough for another developer to use the code in his or her own component or application.

Obviously, if the final product is a component or library of components, the API documentation can be regarded as its user manual. But for other projects the API documentation may not describe how to use the product. In that case a SUM should be written for the end-users. A SUM describes how to use the software. It educates the user what the software does and instructs how to do it. The SUM should be targeted for both novice and expert users.

The SUM may be divided into several sections, aimed at different users, or different levels of users of the product (end-user, operator, administrator, and so on). The SUM can be either an online (context sensitive) help system, a hypertext document, or paper booklet, whatever is best for the type of software. The process does not prescribe a specific table of contents for a SUM, but the following guidelines summarize the issues that may be described in the document:

- An overview of the purpose of the software. The main tasks that the software can fulfill should be described. The prerequisites for using the software may be formulated, if applicable.
- Tutorials for the main tasks of the software. Aimed at novice users, several tutorials may be included to instruct how to accomplish the most frequently used functionality of the software. These instructions may be a solution for a problem formulated in the form of a question.
- A reference section describing all functionality of the software. All the tasks that a user can accomplish with the software should be described in the reference section, including an outline of instructions for the user.
- Summary of errors and their resolution. The frequently made mistakes by users should be described including what the user did wrong and how he or she can solve the problem.
- Examples of use or use cases. To clarify some purposes of the system, an example or use case my be provided for the end-user.

Writing a good user manual is very difficult. Especially when a manual is written by the software's author(s), it can be hard to understand or may even be obscure. The information in this section will not guarantee a clear and well written user manual, it merely provides very basic guidelines as a start. Many books and tutorials are available for writing better English and to aid writing a user manual or assist writing clear API documentation.

4.6.3 Review and Management Practices

This phase may lead to changes in the design or changes in the user requirements. In the case that the design must be updated, the design phase may be reiterated, with the restriction that the team members agree on the changes. Changes in the user requirements must also be agreed upon by the customer. The exploration phase, and if necessary, also the design phase is reiterated.

The source code must be under revision control at all times. Each publicly released version (starting from version 1.0.0) must be tagged. The source code and API documentation should be under review by the development team, a team member may inspect source code of other members from time to time and inform the author(s) of any discrepancies. The pair-programming technique may be applied for this purpose.

A product version number is formatted as *x.y.z*, where *x* is called the major version number, *y* is called minor version number and *z* is called the release version number. When the requirements or design drastically change or major features are added, the major version number is increased. Minor changes to the software will lead to an increase of the minor version number. Bugfixes or patches will have an increase of release version number *z*. Other changes that do not lead to a change in functionality of the product (for example a correction of a spelling error in API documentation) may result in a new deployment release, denoted with a suffix *–rn*. The version number is formatted as *x.y.z–rn* in this case. Here *n* denotes the *n*-th deployment release of version *x.y.z*.

If a SUM is written it should also be under version management continuously. In case the SUM is integrated with the source code, it will be under version control already and the version number of the source code will apply to the SUM as well. When the SUM is a separate (paper) document, the same review and management practices as for the URD, SSD, and SDD should be applied to the SUM. If possible a DSS should be inserted between the table of contents and other content in this case.

4.7 Maintenance Phase

The maintenance phase is entered, when all requirements with the highest priority in the URD are implemented. These requirements are marked as such in the URD. It is not necessarily true, that all requirements are implemented when the maintenance phase starts, for it is possible that some requirements have a priority that indicates that these requirements do not have to be implemented for the software to be deployed. In the maintenance phase the final version of the software product is delivered. Additionally, the phase consist of communicating defects by the end-users to the developers. The developers have to respond by taking the appropriate actions.

Bugs and issues that surface while the software is in use need to be collected and archived somewhere. This is called bug- and issuetracking. Not only annoyances, errors, or other problems, but also missing features are captured in the issuetracking system. For the bugtracking system a competent tool should be used. A priority is assigned to each issue which allows the development team to decide on the necessity to construct and deploy a new release of the software again.

The data from the issuetracking system can be used to decide whether the project needs to go back to an earlier phase – the exploration, design, or construction phase. When the issues only comprise minor bugs and fixes, the construction phase can be entered again, without restrictions.

If features that need design changes are to be added, the design phase can be entered again, with the same restrictions as in the construction phase: the team members must agree upon the design change. When changes in the user requirements are necessary, the exploration, and if necessary, also the design phase can be reiterated with the same restrictions as in the construction phase: the customer must agree upon the changes in the user requirements and the team members must agree upon the design changes. When the product does not fulfill its tasks anymore, and needs a complete redesign, the project can be restarted, and the exploration phase is entered again, using the data from the issuetracker as input.

The issuetracking system may be made accessible for the end-users if the system allows the end-users to use it (in other words, if the users are competent enough or the system is easy enough to use). Otherwise another form of communicating the end-user's issues is necessary and some filter is needed to get the issues into the system — for example a special email address and somebody that reads, processes and answers the emails received at the specific email address, or a special phone-in help-desk facility.

The maintenance phase ends, when the project dies. A project dies when the customer does not report new features or defects anymore, and all requirements in the URD are implemented. This rarely happens. In practice, most software systems require maintenance until the customer stops using the system. That is, when the customer goes out of business or when the system is superseded by another.

4.8 Planning a Project

The matters of planning a project have not yet been researched into full detail. The process was only applied to a small number of projects. In addition, the period of time since the process has been introduced, is not long enough to thoroughly analyze how to plan a project, and there are no results of planning a project in various ways. Further research of literature on the topic, and practical research results are necessary, before conclusions can be drawn.

Although no guidelines to plan a project have been introduced yet, a form of planning a project has evolved in the process of adapting the method described in this chapter. At start of the exploration phase, a date is set for the internal approved version of the URD. When the design phase is entered, a date is set for the internal approved version of the SSD or SDD. Whenever a document — URD, SSD, or SDD — needs to be changed, a date is set for the changes to be ready.

The construction phase is divided into iterations, concluded by a preliminary release of the product. Each iteration is planned by stating what requirements will be implemented in the next release and the duration of the iteration is estimated. The final iteration of the construction phase ends with the final, public release of the product. The maintenance phase is planned similarly, by estimating the time needed to fix a specified number of issues for the next maintenance release of the product.

4.9 Applicability

The proposed process is originally designed for developing components, which does not mean that the process is restricted to the design and implementation of components. We have already applied

it to design and implement two applications as well. Choosing a methodology and process depends on several factors, described in Section 2.7. By applying Cockburn's principles [Cockburn, 2000] on the component process, the following guidelines are derived that reflect the applicability of the process:

- **Principle 1** – The process is designed for small development teams and successfully applied to projects with one up to four team members. Depending on the size of the project, the team size can be enlarged, but there have not yet been any tests applying the process to a project with more than four team members. As already mentioned in the process overview, the process is designed for small projects and small team sizes.
- **Principle 2** – The developed components do not have loss of life or loss of irreplaceable money as a consequence of failure. Because the components are mainly developed for use in an administrative application, failure results in loss of discretionary money or loss of comfort. The process must not be applied to projects with a higher criticality.
- **Principle 3** – The process is designed with cost minimization in mind. Actual return on investment is low, because the components are mainly used to replace outdated code. However, cost minimization must not affect stability and correctness of the components.
- **Principle 4** – The method is designed for co-located teams, or teams that meet on a regular basis. Adaptions to the process are needed for it to work with differentiated teams.

The applicability of the ESA lite standard very much reflects the applicability of the standard proposed in this chapter, as the process is a derivative of ESA lite. Generally the standard works successfully for non critical small projects — less than 15.000 lines of code, excluding comments — with a maximum of five co-located team members.

4.10 Conclusion

The chapter is concluded by arguing that we modified and promoted ESA lite to an agile process by reviewing both Table 2.1 and the principles from the Agile Manifesto from Section 2.5.2. Table 2.1 illustrates the important differences between agile and heavyweight processes. The following list shows that the proposed process has the properties of an agile methodology.

- **Approach** – The process' approach tends to be more adaptive than predictive. Requirements are gathered in advance, but are certainly not fixed. At any time these requirements can be changed, without restriction.
- **Success measurement** – A component's success is measured to the extent it contributes to the existing software: its business value. An indication of how well a specific component contributes, can, for example, be measured by the number of bugs that are reported in that part of the application in which the component is implemented.
- **Project size** – Small, ranging from 2,000 to approximately 10,000 Source Lines Of Code (SLOC).
- **Management style** – Teams are self-organized: a decentralized approach.
- **Perspective to change** – Phases may be iterated arbitrarily, without restrictions. Changes are taken for granted and the process anticipates changes.

- **Culture** – Collaboration.
- **Documentation** – Low, ranging from 3 to 11 pages per 1,000 SLOC, or the other way around ranging from 87 to 383 SLOC per page documentation.
- **Emphasis** – Process is adapted to projects and teams.
- **Cycles** – Numerous: arbitrary iteration of phases.
- **Domain** – Mix of prediction and exploration. The component is designed to fit into the software, but requirements and design may and most likely will change as exploration continues.
- **Team size** – The team size is small and variable and varies between 1 and 4 team members.
- **Upfront plan** – Minimal.
- **Return on investment** – Starting at initial release.
- **Chain of command** – Customer and developer interact directly, there are no persons between customer and developers.

Furthermore, all principles of the *Agile Manifesto*, see Section 2.5.2, are adhered to. The following list enumerates process adherence for each principle:

1. Customers are satisfied by early and continues delivery of valuable software by adhering the open source paradigm to release early and release often. A component is released as soon as a first working version is ready in the construction phase. Every time significant new features are added or bugs are fixed, the product is released again.
2. Changes in user requirements are welcome by the possibility of falling back to the exploration phase. Requirements often change after a release of a test version of the software, which results in a new iteration through exploration and design phase.
3. Working software is delivered frequently, see principle 1.
4. There is no separation of management and development: management is part of the process. The customer is always available; whenever needed the customer can be contacted.
5. Tasks are divided based on preferences of team members and not by top down decisions. Motivation and trust are high, the project teams are operating technically on their own.
6. Face-to-face communication is most frequently used, because the teams are co-located or meet on a regular basis.
7. Working software is the primary measure of progress; a project becomes interesting for the customer only when there is a working version of the component.
8. A constant pace is maintained, because there are no time consuming obstacles to go from one phase (back) to another. Close collaboration within and between teams promotes sustainable development.
9. Team members must be well educated and are recommended to stay up to date with furtherings in the discipline. Constant reviews and collaboration are used as techniques to improve technical excellence and good design.
10. Techniques from principle 9 are used to promote simplicity too.
11. The teams are self-organized and operate technically on their own.

12. All employees regularly meet and reflect on the current methods, tools and projects. The process, tools and projects are improved wherever and whenever possible.

Although the previous arguments claim that the process is agile, it can be argued that a division of a process into phases, a requirements document, and a design document are not common in agile methods. The process' phases are a guideline, for the developer, but also an outline for describing the process. The phases may be viewed as name-tags, to be able to better express what the development team is working on, in some stage of a project. The phases itself impose no restrictions whatsoever on a development team to accomplish their task.

The requirements document is not a contractual agreement between the customer and the development team, which is common in heavyweight standards. It should be a slim document, which can be easily updated, that should serve as a quick reference into the functionality of the software, for the developers to pick out tasks for them to do. The design document contains a decomposition into subcomponents and a subcomponent's essential class diagrams, including a short explanation of those diagrams. This document is useful for two purposes: (1) it acts as a means of dividing the work and debating the design with the development team members, and (2) it provides — in case of a component — a *floor plan* to investigate whether and how well it fits into the existing software.

The following statement from the Agile Manifesto website summarizes the ideology of the process described in this chapter: "The Agile movement is not anti-methodology, in fact, many of us want to restore credibility to the word methodology. We want to restore a balance. We embrace modeling, but not in order to file some diagram in a dusty corporate repository. We embrace documentation, but not hundreds of pages of never-maintained and rarely-used tomes. We plan, but recognize the limits of planning in a turbulent environment" [Beck *et al.*, 2001].

To be able to deploy an agile method, as sketched above, there is a need for a different kind of employees than in heavyweight methods. Namely:

- Employees must be able to do several types of activities, such as management, customer interaction, design, and programming.
- Employees must be able to work with a large number of tools and techniques.
- Employees must be able to quickly switch between tools and techniques.

Deploying an agile method seems to be cheaper and more efficient, but has as a — hidden — cost the need for highly trained and motivated employees. This is not the case in more classical, heavyweight methods.

The overhead costs that were caused by applying a heavyweight process, were related mostly to the context where the process was applied. The final conclusion is that a more agile approach is, in this specific component development context, more effective and more productive than a full fledged heavyweight process.

Chapter 5

An Abstract Model to Select Software Development Tools

The human mind has first to construct forms, independently, before we can find them in things
— Albert Einstein

Chapter 3 already discussed the difficulties to select appropriate tools for a given project. Especially from the ever-widening set of open source and other software tools. An abstract model to assist in the selection process is proposed in this chapter.

5.1 Introduction

In Chapter 3, the importance of tools and selection of the tools and workbench was discussed. There was a felt need for more order in the selection process. After a couple of brainstorming sessions the authors proposed an approach that may be followed in managing the selection process. The approach was applied retrospectively to two case studies. The aim of this chapter is to present the approach. The proposal is an abstract model that promotes careful consideration and transparent documentation of the rationale behind selecting a set of software development tools.

At first reading, the model may appear to be nothing more than common sense – using a general decision-making process that may be applicable in many different areas. For this reason, practical examples from industry are examined in order to shed light on aspects of applying the model to choosing tools for two specific contexts. In both industry examples open source tools are preferred over proprietary tools. The approach is however not only suitable for open source tools but also for proprietary tools, or a mix. Although the logic behind the model is simple enough, the real effort is to discern the appropriate properties of tools being evaluated, and the criteria for choosing between them. Sharing this experience is valuable in itself.

The chapter is structured as follows. A brief overview of previous work related to this con-

tribution is provided in Section 5.2. In Section 5.3 the abstract model itself is described. After that, the two case studies are presented, in Sections 5.4.1 and 5.4.2 respectively. Finally, some conclusions are given and further work is proposed Section 5.5.

5.2 Related Work

CASE tool evaluation has been assessed by several studies, for example the impact on systems development process by Jankowski [Jankowski, 1997], a framework for assessing CASE usage by Sharma and Rai [Sharma and Rai, 2000], CASE tool evaluation at Nokia by Maccari and Riva [Maccari and Riva, 2000] and a method for CASE tool evaluation by du Plessis [Plessis, 1993]. However, all these approaches are targeted at evaluation per CASE tool. The proposed approach compares a set of tools to select a subset for a predefined purpose.

Le Blanc and Korn proposed a phased approach to evaluate CASE tools and indicate how the final selection for a CASE tool is made [LeBlanc and Korn, 1994]. The writers define three phases to be traversed during the process of selecting an appropriate CASE tool. In phase 1, prospective candidates are screened and a small list of CASE tools is drawn up. In this phase a preliminary screening is made by looking at uncommonly provided functionality. In phase 2, a CASE tool candidate is selected that best suits the system development requirements. This can of course only be done when a suitable CASE tool exists for this particular environment. Phase 3 is the actual match between the user requirements and the features of the selected CASE tool. We propose a similar approach, by using a matrix to visualize the reasoning, for choosing software development tools.

The study of CASE tool adoption by Iivari, already discussed in Chapter 3, found that introduction of CASE tools by top-down decision from upper management result in shelfware [Iivari, 1996]. Research by Stobart and associates, and Sharma and Rai show similar results [Stobart *et al.*, 1993, Sharma and Rai, 2000]. Therefore, this approach promotes the decision to be at the developer level, and not at management level. However, in the case of expensive proprietary tools, this must necessarily be a preliminary choice, after which management should be engaged to negotiate about affordability issues.

The definition of *tool* from Chapter 3: “a computer program that helps software developers to create or maintain other programs” applies to the term tool used for the approach in this chapter as well. As mentioned in Chapter 3 this term includes CASE tools by definition. No distinction is made between classes of tools.

An attempt to define practices for choosing and evaluating software engineering methods and tools, is the DESMET project [Kitchenham *et al.*, 1997]. The DESMET evaluation methodology separates evaluation exercises into quantitative and qualitative evaluations. A quantitative (objective) evaluation exercise is aimed at establishing measurable effects. The qualitative (subjective) evaluation is aimed at establishing the appropriateness of a tool or method. In other words, to what extend does a tool or method fulfill the needs of an organization? “The appropriateness of a method or tool is usually assessed in terms of the features provided by the method or tool, the characteristics of its supplier and its training requirements.” A qualitative evaluation is often based on the personal opinion of the evaluator.

The model we propose in the next section is based on such qualitative analysis of the tools

and properties of a tool. The evaluation of a tool is based on literature describing the tool and, if applicable, based on experience with that tool. The proposed model is a visualization of an evaluation of several tools.

5.3 The Model

In this section the abstract model is proposed to aid in choosing a set of tools. It is abstract in the sense that one may choose to implement the principles espoused in many different ways. The model consists of 5 steps, which can be executed iteratively, skipping some if appropriate in the given practical context. By executing the steps, a so called *tool matrix* is constructed. This matrix documents the compliance of tools being evaluated to stipulated requirements, and aids in choosing the set that complies best.

5.3.1 Application

The model is designed to be applied in diverse software engineering methodologies. For example, in a rigorous method where the phases are defined to be followed sequentially — the ESA Software Engineering Standard [European Space Agency, 1991] and Fusion [Coleman *et al.*, 1994] for example — the steps of the model can be executed at the start of each phase. In addition, the selection of tools can be revised by iterating the model steps throughout a phase. In Agile development methodologies [Abrahamsson *et al.*, 2002, Auer and Miller, 2003], the model can be applied equally well. In such a methodology, where progress is designed to be incremental through iterations, the model steps can be followed from time to time to make sure that the set of tools being used still suffices.

5.3.2 Tool Matrix

The tool matrix is a visualization of the tool evaluation and selection process. The columns of the matrix contain the required properties of tools, and the rows of the matrix list the tools and their compliance to those properties. As part of defining the need for a specific requirement, a criticality is assigned to each of the required properties. These criticalities are used as criteria in choosing the final set of tools. The matrix evolves by iterating the steps of the model.

5.3.3 The Steps

Step 1. *Find or refine desired categories of tool support*

The first step in the process of selecting a set of tools is to specify the desired categories of tool support. The actions to take in this step depend on the status of the project. An initial list of desired tool categories is needed, if the project is in its beginning stages. As such, certain tools will be needed, for example a documentation system. As a result, a system to store those documents will be needed too. Often developers have experience in the most common shortcomings that have been encountered on previous projects. This is helpful in extending the list.

It is however unusual to find all desired categories at once. Finding and refining categories of desired tool support is a recurring process that should be repeated throughout the development stages. If the project is already down the path of development, the question is whether the set of currently used tools suffices. As development progresses, new types of support are needed, developers become more familiar with the tools, the tools are improved, and other tools become available. Each of these may lead to a new perspective on the type or degree of tool support needed and its availability. This will lead to a refinement of the current list.

Shortcomings in the current tool support can also be found by researching existing tools. Their usefulness for a particular development process can be appraised by studying documentation, consulting experience reports and experimenting with evaluation copies. Open source tools are ideal for experimenting, as they are freely available and easily found. Collaborative development environments like SourceForge [OSDN, 2004b] and especially Tigris [Collabnet, Inc., 2004] — which focuses on open source software engineering – provide access to tools and corresponding project information [Robbins, 2003]. A variety of open source applications, including useful tools, are available via websites like Freshmeat [OSDN, 2004a].

As a start to identifying necessary categories of tools, the list of commonly used tools from Section 3.4 may be useful. Step 1 is finished when all required tool support categories have been identified.

Step 2. *Specify the required tool properties and their criticality in the tool matrix*

Once the desired tool support for the development process is known, the identified categories must be translated into desired tool properties. These may be in the form of desired features, but should also include quality attributes — such as usability, learning curve or platform support. These requirements are of course personal, but then a tool matrix is for personal decision support. It is also important to note that most tools will have more properties than those that are listed in the matrix. The matrix should however only contain properties that are relevant to the project.

The criticality of a property defines a measure of its perceived necessity. In the model, three levels of criticality are defined:

- **H** – High criticality. A property indicating a highly desired feature. The purpose of the model is to find a set of tools that together will support at least all required properties with this criticality.
- **M** – Medium criticality. A property with this criticality is desired, but tools that do not have support for this property may also suffice.
- **L** – Low criticality. A property that can determine the choice of a tool, when no decision can be made based on the properties with higher criticality.

These levels do not have to be so discrete. Any desired grading can be chosen in a practical implementation of the model. For example, an integer value between 0 and 5 can be used for a more fine-grained scale.

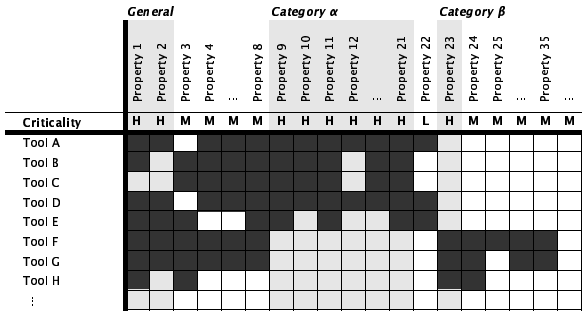


Figure 5.1: A completely populated tool matrix.

Step 3. *Populate the rows of the tool matrix with tools and its cells with indications of compliance*

When all required properties have been specified, the rows of the tool matrix are populated with tools and their compliance to those properties. The most important consideration here is which tools are selected to be part of the matrix, and what this selection is based on. Developers may have experience with previously used tools, and that can be useful. The internet can be searched for tools — especially for open source tools [OSDN, 2004b, OSDN, 2004a, Collabnet, Inc., 2004]. To search for tools in a specific category, the list in step 1 may be useful.

The cells of the matrix are populated for all the tools, by identifying the properties supported by a tool, and specifying its compliance in that cell. In this paper, we have used a simple Yes / No compliance measure — translating into black and white blocks in the diagrams — but also a more scaled measure of compliance may, and should, be adopted.

One or more features of a tool might be found that translate into a desired property that is not yet listed in the matrix. In this case, step 2 needs to be executed again, by adding the property to the matrix and defining its criticality.

Step 3 is finished when the compliance of all tools to the desired properties has been identified, and when no new properties or tools can be added. An example of a completely populated tool matrix is depicted in Figure 5.1.

Step 4. *Analyze the tool matrix*

Once all required properties and tools are listed and the appropriate cells of the tool matrix have been populated, the matrix can be used to select the actual tools that *could* be used. Before the final selection of tools is made, the tool matrix must be analyzed as follows: If there is a critical property that is not covered by a single tool, which can be identified by an empty column in the tool matrix, three cases can be distinguished:

1. The tools have not been examined well enough, and a property of a tool already in the matrix has been missed. The property is thus supported after all. This type of problem can be avoided by showing the matrix to experienced tool users and evaluating the tools more thoroughly.
2. Maybe there are tools available that support the property, but none of them are listed in the matrix, because the tools were not found. Go back to step 3, try searching again, and include

the tools in the matrix if found.

3. No known tool supports functionality for the required property. Tools which may support the desired property but are not being considered for some other reason should be included in the matrix, with their undesirable properties showing why they have not been selected. It is important to document both the positive and negative decisions.

   In this case, if no tool is found to support a critical property? There are three possibilities:

   3.1 Create a custom tool to support the unsupported property. The tool can either be developed internally, or someone else can be contracted to develop it. This decision must be made bearing in mind several factors, including the cost in time and money to develop and support the tool, and company motivation for such involvement.
   3.2 Add the feature to an existing tool. Again this can either be done internally or a request can be made to have the feature included in a next release of that tool.
   3.3 Reassess the importance of the required property. This may involve decreasing the property criticality or dropping the requirement entirely.

   At this point, the process can be continued.

Step 5. *Select a set of tools*

In this step, an optimum set of tools is selected by going through the matrix. This set should of course cover all of the critical properties. The less critical properties can be important to make the decision if all the highly critical properties are supported in more than one tool. Choices among alternatives may however be dictated by personal preferences. For example, you may prefer an integrated development environment, or a collection of smaller, more task-oriented tools. Coming to a decision may also force you to consider several additional factors. For example:

- Are there developers that have experience with the tool?
- Is the tool easy to use — does it have an intuitive graphical user interface?
- How well is the tool supported — is it still maintained, is it in beta stage or a stable version, is the tool available for the desired platform and so on?
- How much does it cost to buy a license or use the tool?
- How difficult is it to learn the tool — in terms of man-hours?
- How well does the tool integrate into the existing set of tools?

It is difficult to imagine one ideal algorithm to determine the final choice of tools that will be used. The matrix provides a detailed trace of what is important in that choice, and the suitability of the candidates.

Many of the previously mentioned factors could also be added to the matrix, refining the desired properties. The scope of what is included (and what is not included) in the matrix as well as the determining the weight of the factors in the final decision is up to the decision makers. The matrix gives them a reasoned path up to that point.

5.4 Case Studies

In the following sections, the retrospective application of the model in two case studies is described. The case study in Section 5.4.1 involves the tool environment of the component development process. After introduction of the component development tool environment, we were asked to introduce a version system and build system for the administrative application as well. In Section 5.4.2 the choice for these tools is validated with the tool matrix. Both case studies give an indication how the use of the model exposes both the strengths and the weaknesses in various tool choices.

5.4.1 Case Study I: The Components

The discussion below indicates how the abstract model from section 5.3 was applied after the fact to eight different tool categories used in the component development process. This is not an exhaustive list of tools used, but adequately illustrates the construction and use of the model. Since the case study illustrates the use of the model at a later iteration of the software process, the desired category of tool support was already known at the time of its application. As a result, it was not necessary to carry out the first step recommended in Section 5.3.3.

The second step in Section 5.3.3, *specify the required tool properties and their criticality,* was perhaps the most challenging part of the case study. Articulating what should constitute a list of desired properties proved to be quite challenging and evoked much debate, the details of which will not be recounted further in the narrative to follow. The eventual set of properties used and their criticality assignments represent our consensus view reflecting our perceived priorities and preferences in a particular context. Future applications of the model in other contexts could perhaps reuse some of these properties, but their appropriateness should be thoroughly re-assessed. Thus, for example, the authors' preference for open source tools should not be construed as a universal endorsement for open source in all contexts. The widely acknowledged potential hidden costs associated with open source (especially in regard to limited skills) is an issue that is orthogonal to the application of the model, and should be incorporated as a property in the model if relevant to a given context.

The third step in Section 5.3.3, in which *the rows of the tool matrix are populated and the appropriate cells are filled in,* was not too difficult, since we were well aware of many alternative tools for the tasks that they had been undertaking. Nevertheless, applying the step did involve a number of web searches to discover new tools that might have become available, and also, in certain cases, to verify or discover whether particular products had a required property. In this regard, the version system is an important example where a new system was discovered that could replace the current.

The following subsections walk through the eight different tool category sub-matrices, recounting some of the analysis, insights and conclusions elicited by their construction. These subsections therefore focus on the fourth and fifth steps of Section 5.3.3.

For completeness, the initial overall tool matrix for the eight tool categories is shown in Figure 5.2. Note that the first nine columns enumerate general properties that were considered to be relevant to all categories. Associated criticality values are also specified. However, the criticality values in this figure are nominal: in the more detailed analysis reflected in later figures they differ

from one tool category to the next. Also, some of the later figures include additional category-specific properties that are not displayed in Figure 5.2. It was only during the analysis step that the need for these properties was perceived. This illustrates the iterative nature of the model's use and development.

Compiler

As the components were to be used in a Delphi application, Borland Delphi had to be used to compile them. There are several flavors of Delphi, and three of them are listed in the matrix in Figure 5.3. Properties in the matrix reflect the characteristics of the development process. For example, the development process incorporated an automated build process which, in turn, required command line access to compiler functionality — hence a high criticality to the property requiring a command line interface. In addition, Windows was required by the application developers, though Unix was our preferred operating system. The criticality levels for these properties reflect these needs.

Entries in the first three columns indicate that the only way to support all three of these properties is to use a combination of the three tools being evaluated. This was, in fact, done. The Borland Delphi Enterprise IDE was used as the main developer environment. To support builds from the command line — for the build system — the command line compiler was used. And, since the components also had to be be usable in Unix, Kylix was used for that platform.

Documenting- and modeling tools

In order to specify what a component is supposed to do, documents were created to contractually specify the requirements of a component and to illustrate its design. To this end, a documenting system and modeling tool were needed. The tool matrix in Figure 5.4 summarizes the important properties for these two tool categories and lists the documenting systems and modeling tools that were investigated.

The initial choice for documenting and modeling tools was not based on a tool matrix. Instead, it had been decided to use Microsoft Word both for documenting and for modeling. The tool matrix highlights the weakness of this choice. For example, Word files are stored in a binary format, which makes it difficult to automatically merge a document that is changed by more than one developer concurrently — depicted by the plain text input format property. It was felt that Word does not supply a convenient mechanism for adding references in a document and there is no direct support for UML in Word so that creating design models became tedious and time-consuming.

Our relative independence (we are located remotely from the application developers) allowed us to choose alternative documenting- and modeling tools to Word, which was the tool that the application developers continued to use. We decided that Unix support was more important than having a GUI, and selected LaTeX together with BibTeX as documenting system. However, the table indicates that MikTeX provides all general and documenting system properties that have a high criticality level, but on a Windows platform. Furthermore, it includes a graphical user interface. In fact, it is a port of LaTeX, BibTeX and related tools to Windows, and would have been chosen if we were obliged to work in a Windows environment.

Xfig was the first modeling tool that was used. Subsequently it was decided to change to

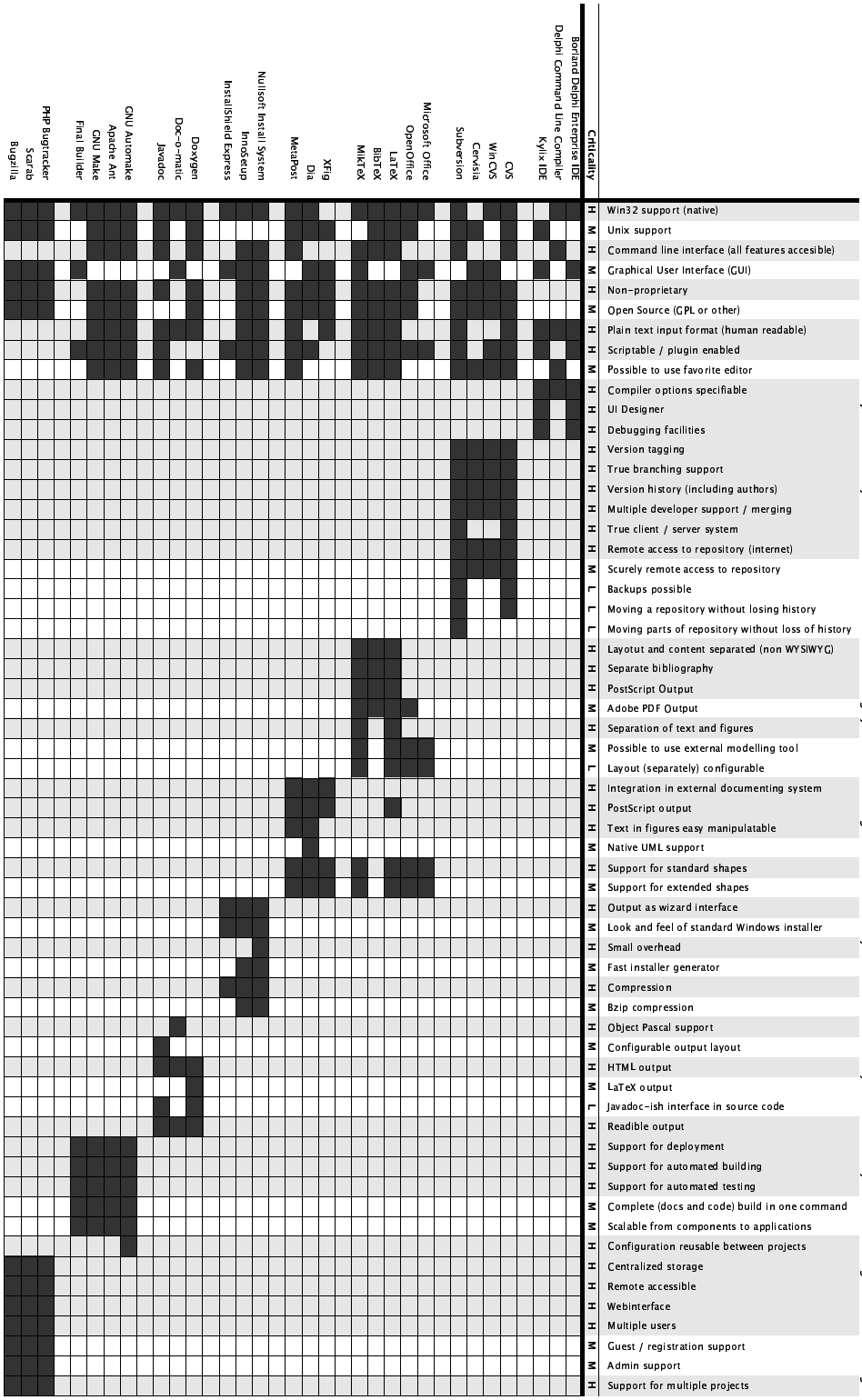


Figure 5.2: Case study 1 matrix.

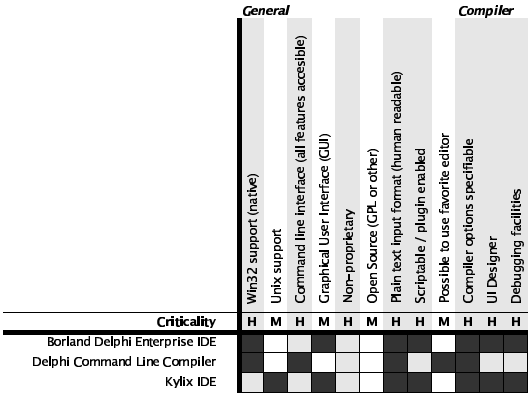


Figure 5.3: Compilers.

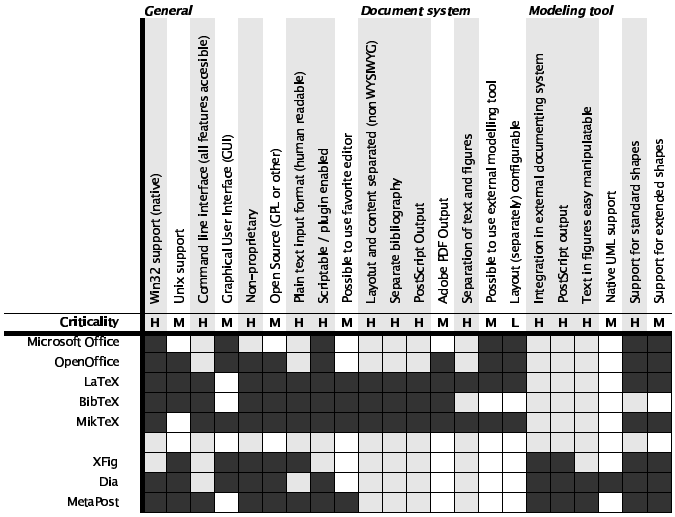


Figure 5.4: Documenting- and modeling tools.

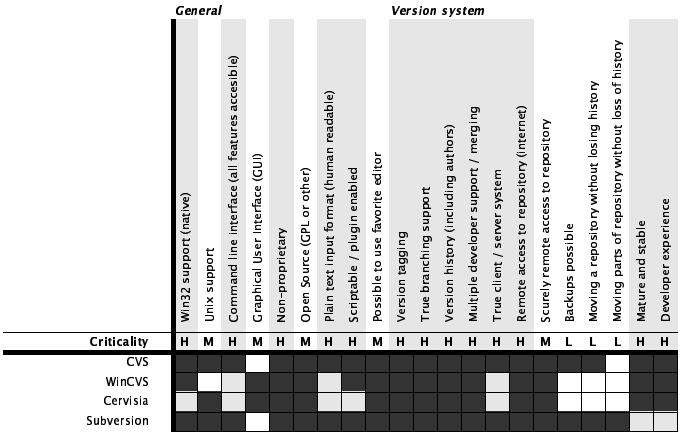


Figure 5.5: Version systems.

Dia. The matrix indicates that text in figures is easy to manipulate on both the Dia and MetaPost systems. Thus it is easy to change property, method and procedure entries in a UML class diagram. However, Dia does not support plain text input format and does not provide a command line interface. To this extent, the tool matrix highlights the fact that MetaPost should have been the preferred system.

Indeed, there was even a greater incentive to switch over to the MetaPost system, since experience with Dia showed that it was not very stable. (This suggests that *stability* should be considered as a property in future iterations of the modeling exercise). However, there is a built-in inertia in changing from one tool to the next. In the case of switching modeling tools, the inertia is particularly severe, because all existing images have to be converted to the new system. Nevertheless after tackling the conversion, we have now switched to MetaPost successfully.

Version system

At first glance, the overall tool matrix in Figure 5.2 might suggest that the best version system choice is obvious: according to that figure, Subversion provides support for most properties. However, the first stable release of Subversion was only available at the time of constructing the model. No such release was available when the decision for a version system had to be made. Fortunately, CVS also provides for all critical properties. In addition, most developers were already familiar with CVS.

Subversion and CVS do not provide a graphical user interface themselves, but there are third party GUI front-ends available for both version systems. Front-ends for CVS are WinCVS on the Windows platform and Cervisia on Unix platforms. The widespread use of CVS made it stable and secure. Maturity and stability are particularly important in a version system tool. When these properties are included in the matrix, as in Figure 5.5, CVS emerges as the most appropriate choice. Nevertheless, the matrix highlights the fact that Subversion seems very promising; it provides support for all critical and non-critical properties and it is possible to migrate a CVS repository to Subversion. For this reason, it could be considered as a tool once it has reached stability.

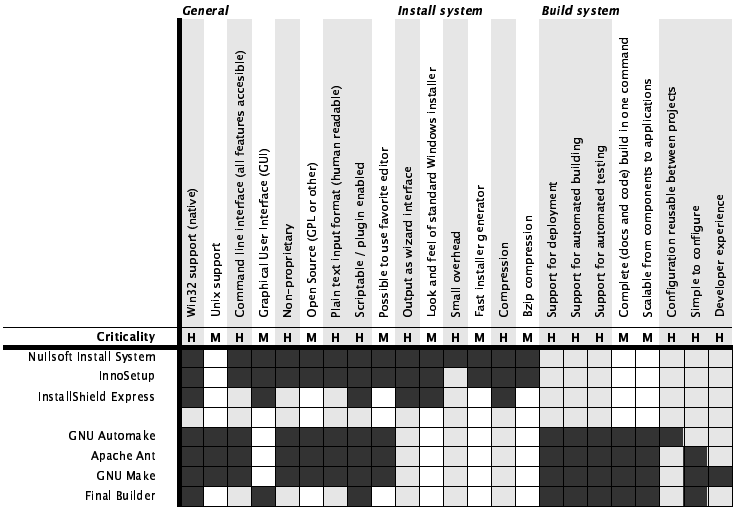


Figure 5.6: Install- and build systems.

Install- and build systems

The matrix in Figure 5.6 shows why the choice of an install or deployment system was quite easy. The Nullsoft Install System is the only system to implement all required properties. InstallShield Express, a proprietary tool, is also included in the matrix. It is important to note that the choice of properties is most important — a different set easily results in a different choice. If the required properties were chosen differently, InstallShield Express could have come out as the tool that would fit the needs best.

On the other hand, we had difficulty in choosing a build system. Because the individual component development projects all have more or less the same setup – they consist of several documents and several Delphi packages — the build system had to be reusable from one component development project to the next. Figure 5.6 depicts four investigated build systems. They all have more or less the same properties. Since Final Builder is proprietary, and since open source tools were strongly preferred, three alternatives remain. Only GNU's automake suite provides a way of setting up a build for a generic project and then reusing it by instantiating that setup for each project.

However, GNU automake would be hard to configure for the component development process, because it uses the M4 macro language and is designed specifically for Unix software. A new set of macro's would have to be written for the applications used in the build process. Moreover, we were not familiar with the language in which this would have to be done. The matrix shows that the *simple to configure* property was deemed to have a high criticality. As a result, the matrix confirms that none of the systems are both easy to configure and allow for a configuration that is reusable between projects. To this extent, it was decided to develop a customized build system, which is described in detail in section 6.2.1.

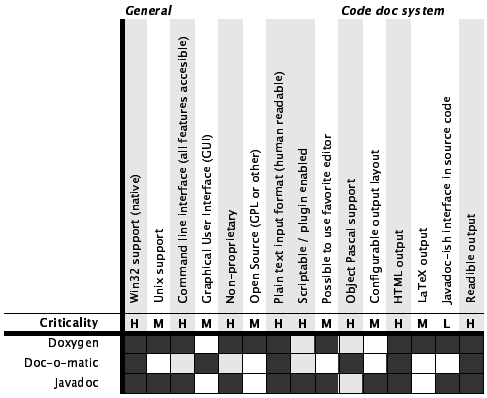


Figure 5.7: Code document systems.

Code Documenting System

The matrix depicted in Figure 5.7 lists the key features that were considered important for a documentation generation system to be used for the components that were being developed. Doc-o-matic is a proprietary tool. In addition to this disadvantage, its interface for producing documentation is disjoint from the interfaces we are already familiar with. Furthermore, the cost of learning it also has to be taken into account. Doxygen could have been an alternative, but for the fact that it lacks support for Object Pascal. In addition, most of the developers did not like the generated output of Doxygen.

We had previously used Javadoc to create Java API documentation in another project. However, Javadoc was not an option as it is designed specifically for Java.

The matrix therefore indicates that no suitable tool could be found that would satisfy all the required properties, which supports the authors' decision to create their own API documentation generation system. The development of this API documentation system is described in Section 6.2.2. The system is called appDelphiDoc, which is a replica of Javadoc that is tailored for Object Pascal. The cost of its development was higher than the cost of buying Doc-o-matic, but the fact that it integrates seamlessly with the other tools is considered to be a major advantage.

Bug- and issue-tracking system

In the overall matrix in Figure 5.2 it is not clear which bug- and issue-tracking system should be chosen. All relevant systems in that matrix support all required properties, except for a command line interface, plain text input and scriptability which were not considered to have a high criticality.

In reconsidering the matter in order to generate Figure 5.8 for the bug- and issue tracker tool matrix, extra properties were added that differentiate between the tools: *easy to install* and *minimalistic*. PHP Bugtracker complies with both these requirements. We in fact originally chose that tool precisely because it was easy to install and because the necessary server with required applications was already up and running. Another advantage of PHP Bug tracker is that it is a simple system. We found that Scarab and Bugzilla were too elaborate for the needs of the project.

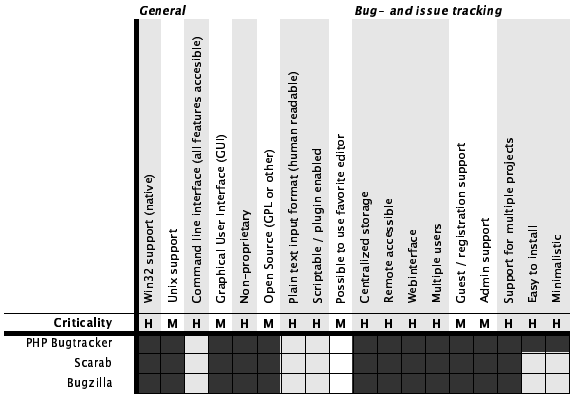


Figure 5.8: Bug- and issue-tracking systems.

5.4.2 Case Study II: The Administrative Application

This section presents a second, smaller case study in which the model was used to validate the choice of tools to enhance the development process of the enterprise administrative application itself. The tool matrix depicted in Figure 5.9 was used to analyze the currently used tools.

Version system

The application developers had been using Microsoft Visual SourceSafe as version system during almost the entire development process. Though they felt that they were missing quite a lot of functionality, they never really tried to introduce a new version system. True branching[1], for example, is not supported in SourceSafe. This meant that bugs would be solved and features would be added in the main trunk of the development tree. As a result, whenever a version that solved a bug was released, new bugs would invariably be introduced into that same release, because it contained new inadequately tested features.

Version tagging and branching support are essential to solve this problem. With version tagging every publicly released version of the product can be marked. Using separate branches, whenever a bug is reported the developers are able to take the latest tagged release and solve the bug in a separate branch and release that version to the client. This way no untested features are included (because the bug is fixed in the latest public release and not in the main development trunk). The bugfix can then be merged[2] into the main development trunk.

As discovered in the previously described case study, Subversion covers all the required properties, but was not stable at the time of the investigation. The matrix indicates that CVS and WinCVS together provide for all the critical properties. As a result, it was decided to switch to the CVS system in combination with the WinCVS front-end.

[1]Branching, in software configuration management, is the duplication of an object under revision control in such a way that the newly created object has initially the same contents as the version branched off from, and —more importantly —development can happen in parallel along both branches.

[2]Merging is the process of copying the differences accrued to an object on another branch to back to the parent branch (usually called main trunk).

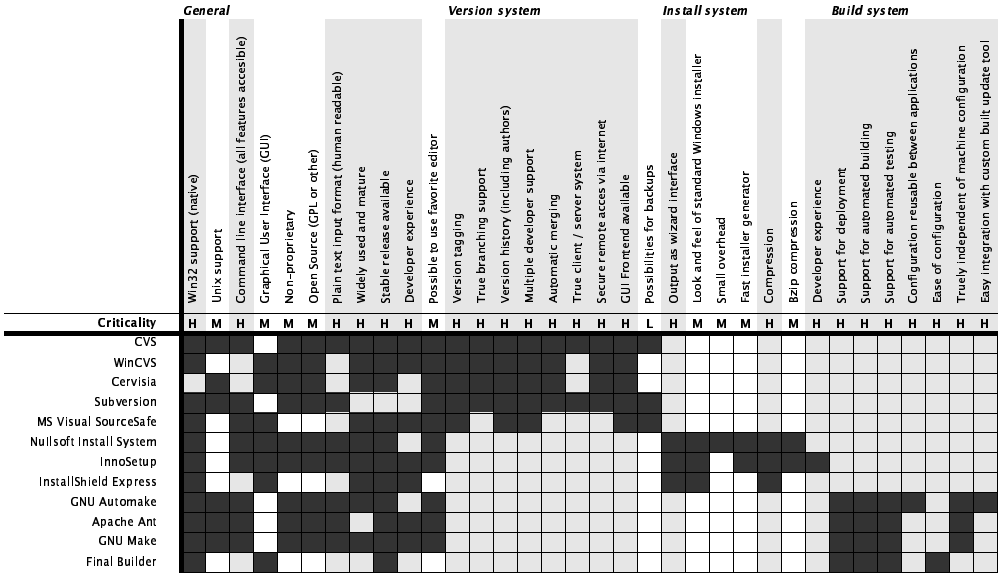


Figure 5.9: Case study 2 matrix.

Autobuild and install system

The developers did not yet have a generic build system. Every developer used to build the software within the IDE and an actual release was built on a separate system using a custom build system. There were problems with that setup because the configuration of both build processes — IDE and build system — differed widely. For example, the IDE statically linked the software, however, the actual release version was dynamically linked. So, the problems of dynamically linking the new features only came to surface when the actual version was due to be released. As a result, the dynamic build process and the actual release sometimes took as long as a complete day.

The tool matrix depicted in Figure 5.9 shows that no tool implements all of the critical properties. A slightly modified version of the component development build system could be used for the application itself. The last column of the matrix shows that support for the custom built update tool needed to be added. Innosetup was used to package the application so logically it would be a wise choice to keep Innosetup and incorporate it into the build system.

The system is currently being used and has significantly improved build times. The time of build and actual release is reduced from an average of 4 hours to an average of 5 minutes. Because there is now only one automated way of building the system, this implies that a release version can be built by any developer who can do a local build.

5.5 Conclusion

Choosing the right set of tools is important. A well-chosen set of tools can cut down on development costs significantly. Open source tools proved to be an important source from which to choose tools. Building a custom tool may well improve the development process too, but one should be careful in making the decision to build a new tool internally.

A model to assist in choosing a set of tools was introduced. On the surface, it looks just like common sense, but when the process of drawing up the matrix is undergone, a lot of issues will surface that would otherwise have never come to mind. Practical industry examples were given to illustrate the usefulness of the abstract model.

In most cases, tool choice relies mainly on experience and gut feel. These can be imprecise and misleading. Drawing up a tool matrix formalizes the reasons for a choice, which can be reviewed or defended. A tool matrix can also highlight reasons for problems in tools already being used.

A tool matrix can be applied for personal use, in the sense that it is instructive just to build up the matrix. It can be used within a group of developers to agree on a set of tools or to convince management to buy a tool that is needed. It can also show that an open source tool fits the needs just as well as — or even better than — some expensive proprietary tool. The tool matrix has many practical purposes; it itself is another tool to be included in a developers toolbox.

Interesting work still needs to be done to support the process that is described. To make it easier for people that are in the process of creating a tool matrix to fill out the required properties, template tables that summarize required properties of a common tool category could be constructed. In these tables no criticalities would be assigned to properties, because criticalities are subjective.

Generic tool matrices listing the properties of available open source tools would also be helpful in the process of drawing up a tool matrix. These tables would need to be revised quite often because tools are constantly being extended or changed, and new tools emerge on a regular basis.

A tool to create a tool matrix could be helpful as there are currently no specialized tools for this purpose. Such a tool could be able to assist in making the decision for the final set of tools, because the tool matrix allows the choice for tools to be deterministic. The previously mentioned template tables that summarize required properties of a tool category can be presented to a user by the tool matrix support application. Combined with schema's of properties of specific tool implementations, the tool matrix support application is able to already narrow down the set of tools. The important part that is left to the user, is filling out the criticalities for the required properties, after which the tool matrix support application can present the final set of tools.

The abstract model introduced in this chapter is specifically designed to choose tools in a software engineering process, but could conceivably be more generally applicable. The way of making a decision, as described by the model, could also be used in a generic approach to documenting the decision making process in other areas.

Chapter 6

Practical Work

program: n.

1. ***A magic spell cast over a computer allowing it to turn ones input into error messages.***
2. ***An exercise in experimental epistemology.***
3. ***A form of art, ostensibly intended for the instruction of computers, which is nevertheless almost inevitably a failure if other programmers can't understand it.***

— Entry from the Jargon file by Eric S. Raymond

This chapter gives the details of the practical work. Chapter 3 highlighted the importance of tools and Chapter 5 proposed a model to choose a set of tools. The latter chapter also showed the results of applying the model in practical cases. This chapter will describe in detail the practical work for the final tool environment. The process from Chapter 4 was applied in practical cases, which are discussed in this chapter as well.

6.1 Introduction

The first case study from Chapter 5 applied the abstract model for selecting software development tools for the component development process. The resulting tool environment needed two additional tools to be built: an API documentation tool and a customizable automated build system. The details of the API documentation tool are described in Section 6.2.2 and the automated build system is described in detail in Section 6.2.1.

The proposed model from Chapter 4 was applied to several projects. We picked two case studies which are discussed in detail in Section 6.3. The first case study, concerning the design and implementation of a collections component, is the more straightforward case. Capturing the requirements proved to be not too difficult, because the demands for the component could be well defined. This case is described in Section 6.3.1.

A less straightforward case is described in Section 6.3.2, where the process was applied in the development of a localization and internationalization component. The specification of the needs for such a component proved not to be easy. The implementation of only a component did not suffice, a support application was necessary. The process was applied for the design and implementation of both the component and the supporting application.

Furthermore, two components to support application security, and encryption and decryption of information, are developed using the process. These two components are developed by two other colleagues, their experiences are described in Section 6.3.3. The component development process was followed too, for the design and implementation of the API documentation tool.

6.2 Tool Environment

Figures 5.6 and 5.7 illustrate the needs for the automated build system and API documentation tool respectively. Both figures were used to conclude that the choice to build custom tools in these cases was right. This section will describe the tool environment and the practical work concerning the construction of the automated build system and the API documentation system.

Using the abstract model applied to the component case study, the selection of tools for the tool environment can be read from the diagrams. Interpreting Figures 5.3 to 5.8, the following tools constitute the current tool environment:

- CVS
- Custom automated build system
- LaTeX
- BibTeX
- MetaPost
- Delphi IDE
- Delphi command line compiler
- Custom API documentation tool
- NSIS
- PHP Bugtracker

From the same figures, the important requirements can be read, for both custom make tools. In an ideal situation, the custom made tool should support all of the critical and non-critical properties for the tool category it belongs to.

One extra tool category needs to be added to the tool environment which is not discussed in Chapter 5: a release publication system. A release publication system provides the means to make a release of a product available to the end-users. Examples are: an FTP server, publicly accessible CVS repository, and a website. The latter example is used for the components; a website served by Apache and PHP provides links to download component installer packages. This is relevant for the automated build system to support deployment.

6.2.1 Automated Build System

Figure 5.6 shows that GNU make already supports all critical and non-critical properties, except a project's configuration will not be reusable between projects. All the Makefiles for one project need to be adopted for another — a tedious job taking a lot of time and effort. However, GNU make may be used if there is some additional tool that simplifies Makefile administration. Using GNU make is preferred, because by using it, a lot of requirements for the automated build system are for free. A tool can simplify building Makefiles by requiring only the input that differentiates between projects and automatically building the Makefiles, which is the main facility of the automated build system.

The command line interface of a tool is the easiest way to communicate from within Makefiles. Will the API documentation tool have a command line interface, then all but the Delphi IDE, PHP Bugtracker and the website (where the components are deployed) are tools with such a command line interface. Interfacing the IDE or the bugtracker is not highly important, however, the website must be updated by the automated build system to support automated deployment. This problem is solved by communicating with include files that are read by PHP to process the links to the packages.

The custom made parts of the autobuild system provide the Makefile configuring layer and deployment support. It was decided that a scripting language would be most useful, and as developer experience with Perl was available, Perl was chosen to implement the autobuild system. Two scripts were written:

- **Configure** – A script that reads a set of project configuration files and creates a set of Makefiles to build the project.
- **Release** – A script that reads the project configuration files and deploys the files that are constructed by building the project.

The variables that make a project unique are stored in the project's configuration files. A project has a top level configuration file identifying and describing the project, a configuration file for each document, and a configuration file for the source code. The configuration files for documents and source code contain revision numbers, the files that must be build, required components, and so on.

After writing the configuration files, a developer can generate the necessary Makefiles to build (a part of) the project. First, Configure will check if the programs that are needed to build the software are present on the system. It will warn or err, when the appropriate executables could not be found. When the targets have been successfully built, they may be deployed. The Release script is written to facilitate deploying (a part of) a project. After checking whether the developer increased the version number and applied the appropriate tags to the CVS repository the script will upload the necessary files to the webserver and modify the webpages by replacing several include files.

Compared to the situation before the automated build system was introduced, build times have significantly improved. Table 6.1 illustrates the mean build times of several parts of three different components before and after the introduction of the automated build system. The build times are measured on an AMD Athlon XP 3000+ system with 1Gb of memory. The time needed for building a document has not decreased much, because the actions needed to build a document by

	Before	**After**
URD	0m33s	0m09s
SSD / SDD	1m10s	0m29s
All documents	1m55s	0m39s
Source code (one Delphi version)	0m48s	0m01s
Source code (all Delphi versions)	2m04s	0m01s
API documentation	0m33s	0m01s
Installer package	3m16s	0m03s
Document deployment	5m19s	0m11s
Package deployment	12m37s	0m18s

Table 6.1: Mean build times before and after introduction of automated build system.

hand, are not much more complicated than building it with the automated build system. When all documents or all source code versions are build, already a significant decrease of build times can be observed. The most significant decrease of build time is achieved in the case of a package build and deployment.

Because the components should be released early and often, the significant decrease in package build and deploy time is an important advantage. Before the introduction of the automated build system, a release of a new version was sometimes postponed because building and deploying the package took too much time. Currently, whatever the reason to release a new version — even when only a spelling error in the API documentation is fixed, for example — a new version can be quickly and easily released. Besides the decrease in build times the automated build system has other major advantages:

- The probability of making mistakes is much lower using an automated system. Obviously, when a person needs to type in only one command instead of ten, it is less likely that he or she makes a mistake.
- The possibility to incorporate checks into the system. To prevent developers from neglecting prescribed actions, the system may halt and instruct the developer which actions must be taken before the operation can be continued.
- No interaction is required with the developer. The system can be run without supervision.
- The automated build system is not designed for a specific operating system. It only depends on having specific tools installed.

There are also some disadvantages. Sometimes it can be more complicated finding an error, because an extra layer — the automated build system — is added. Also, if additional actions are needed to build or release a project, the automated build system has to be adapted. The major disadvantage is certainly the cost of developing the autobuild system. The number of man-hours

needed to build the automated build and release system is roughly 150. The costs seem high, however they are recovered easily within a year. Say the new system saves approximately 15 minutes per developer per day. With four developers, the result is one hour saved each day. After 150 days the man-hours spent on the system are paid back.

Overall, the advantages of the automated build system outweigh the disadvantages. The most important properties of the autobuild system are to assist in preventing, detecting, and localizing errors. The automated build system is an example that reflects the importance of tools, and the importance of choosing the right set of tools described, in Chapters 3 and 5.

6.2.2 API Documentation Tool

The context of the software components imposes the need for a clearly defined interface. The functionality of each class, and its properties and methods should be clearly described and made publicly accessible to the end-users. The popular term for this is *API documentation*. The API documentation of a piece of software is the user manual for the developers that want to use the software. Many modern tools allow developers to write API documentation in their source code using special comment style. This avoids duplication — of specification of types, classes properties and methods — and while reading and writing source code, its explanation is at hand as well.

A very popular tool for managing API documentation is Javadoc. The authors already used Javadoc for previous projects and liked the result. Unfortunately Javadoc works with Java source code only, and we started to look for a similar tool that could be used for Delphi. The only suitable tool we found was Doc-o-matic. Unfortunately Doc-o-matic is a proprietary tool and buying the latest version for four developers would cost $1680,-.

Besides the price, the decisive factor was that integrating the code documenting tool into the process and build system would be much easier when the tool is custom developed. Building the customized API documentation tool allows specifically tailored outputs — a paper document in exactly the same style as the other project documents for example. The automated build system can be tuned to the inputs and outputs of the tool and vice versa. The tool can comply more specifically with special wishes of the developers. These are all advantages of a better integrated customized code documenting system. Figure 5.7 from the case study described in Section 5.4.1, depicts these arguments in the form of a tool matrix.

The process from Chapter 4 prescribes writing a document that describes coding standards. Such document was already written, and before the development of the tool was started, this document needed to be modified. The rules for the special formatted code comment, called *doc comment*, were included in the Coding Standards Document (CSD) [Zwartjes, 2002]. The notation from Javadoc was used as a source of inspiration. Simultaneously, a URD was written for the new project, called appDelphiDoc[1] [Zwartjes, 2003]. The tool's design is described in its SDD [Zwartjes, 2004]. The design is straightforward: a lexicographical scanner, used by an object pascal parser — extended to read doc comment — that produces a parse tree (in memory). A visitor, traversing the parse tree, generates the documentation. The visitor's type determines the type of documentation that is written: Hypertext Markup Language (HTML), LaTeX, or Extensible

[1] A company standard prescribes the project name of an application must be prefixed with *app*, as well as the project name of a component should be prefixed with *doc*.

Project	Number of developers	Total lines of code	Lines of source code	Blank lines	Lines of comment	Percentage of comment	Pages of documentation	Lines of code per page of documentation	Pages of documentation per 1000 lines of code
docSet	2	4,781	2,266	601	1,914	34.4%	26	87	11.3
docTLUM	2	10,827	7,194	1,281	2,352	21.7%	43	167	6.1
appTranslator	1	16,473	9,413	2,018	5,042	30.6%	34	277	3.6
appDelphiDoc	1	12,022	7,278	1,582	3,162	26.3%	19	383	2.6
vizLicense	1	8,183	4,361	1,011	2,811	34.4%	23	190	5.3

Table 6.2: An overview of the statistics of the projects used in the case studies.

Markup Language (XML). Detailed statistical information about this project can be found in Table 6.2.

The tool has been built and integrated into the automated build system successfully. Currently only HTML output is supported. Roughly 300 man-hours have been spent to document, design and implement the tool. The analysis from the automated build system cannot be applied in this case; the recovery of man-hours is not a trivial calculation. The cost of buying a tool is saved, as well as the cost of learning to use that tool. The positive and negative effects of that other tool however remain unknown. In this case, developing a custom tool seems fruitful. The amount of man-hours is acceptable compared to the cost of buying a third party tool. But, as already mentioned in Chapter 5, the necessary caution is in order when the decision of building a custom tool is to be made.

6.3 Case Studies

This section describes three case studies to illustrate the results of applying the process from Chapter 4. The case studies comprise the development of several components and an application. Each

Metric	Value
Number of projects	9
Number of developers	4
Total lines of code	76,283
Lines of source code	44,164
Blank lines	9,189
Lines of comment	22,930
Percentage of comment	30%

Table 6.3: An overview of the statistics of all projects.

case study is presented in its own subsection. Table 6.2 summarizes the statistics for the selection of projects that serve as a case study in this chapter. The previously discussed appDelphiDoc tool is included in this table as well, to give an indication of the size of that project.

The component projects described in the first two case studies were started already in 2002, using the ESA software engineering standard. Both projects were restarted a year later to try out the new process. This restart provided data on cost in man-hours, that will be presented in the following sections. In addition, the benefits and drawbacks of applying the new process in these two case studies are discussed.

The third case study describes the development of security components. These components were designed and implemented by two other employees. The case study summarizes interviews with the employees and shows the man-hours they spent.

An overview of the statistics of the overall set of projects, (including a total number of eight projects) is presented in Table 6.3. The projects are developed by and divided amongst four employees, including the two authors. Table 6.2 indicates the number of developers that worked on the source code of a project. All four employees are TU/e students, and were hired because they are able to work independent and because they are creative and flexible. Chapter 4 already discussed the type of employees that are a prerequisite for a more lightweight process.

Considering Table 6.3, it is also possible to regard all software components to be one combined project with a team of four developers. Within this single project, each employee has its own responsibilities for a specific part of this project. It can be interesting to view all projects as a single large project as well, however, it is not discussed in more detail in this dissertation.

6.3.1 Case Study I: Collections

This section outlines the development of a component that implements collections, the first row in Table 6.2. A real object-oriented approach to collections is not included in the standard libraries of Delphi 5, 6 or 7. Collections are essential in other components and applications, and the need for a well designed object-oriented and extensible implementation was high. A project called docSet

was started to fulfill this need. The specific lifetime management requirements, and the specific requirements for using the component in the internationalization component, were of overriding importance to custom develop a collections component.

Originally, before the introduction of the component development process, we started the project by creating the management documentation, as prescribed by the ESA software engineering standard. This set of documents consists of a Software Project Management Plan (SPMP), a Software Configuration Management Plan (SCMP), a Software Quality Assurance Plan (SQAP), and a Software Verification and Validation Plan (SVVP). Concurrently, while the management documents were written, we arranged a meeting with the customer to discuss about the functional details of the component. The requirements for the component turned out to be straightforward and only two meetings were necessary to gather all requirements. The requirements were written down in the URD. The following list summarizes the types of collections supported by the component, reflected by the requirements from the URD:

- **Bag** – A collection without any restriction. The items in a bag are unordered.
- **Ordered Bag** – Ordered variant of a bag. The order should be defined by the developer.
- **Set** – A mathematical set: a collection that may not contain duplicate items. The content is unordered.
- **Ordered Set** – An ordered version of the mathematical set. The developer should be able to specify the ordering himself.
- **List** – A collection that preserves the order in which the items are added. A list may contain duplicates.
- **Stack** – A Last In First Out (LIFO) stack. Items are added on top of the stack, and retrieved from the top of the stack.
- **Queue** – A First In First Out (FIFO) stack. Items are added to the end of the queue, and retrieved from the head of the queue.
- **Tree set** – A single-rooted tree. The items are added to the root of the tree or to a branch originating from the root. The tree set provides different search mechanisms: depth-first and breadth-first. The tree set structure was added in a later stage of the project.

After the URD was approved by the customer, we started working on the SRD, ADD, and consequently the DDD. The number of man-hours spent per document are detailed in Table 6.4.

By experiment, the source code was written by an external development team of students from the university in Sophia, Bulgaria. All design documents were sent to the development team, and their assignment was to implement the design described in the documents. Every week their source code was sent in and thoroughly reviewed. The resulting review reports were sent back to the development team in Bulgaria. Despite the detailed documentation, it took the development team a lot of man-hours to implement the component and it took a lot of man-hours to review the source code, and write review reports. The limitations of email and a foreign language made communicating with the development team difficult and troublesome. In the end, the results were very disappointing — the source code was of poor quality. The number of man-hours spent on the source code can be found in Table 6.4 too.

The project was restarted by applying the new process. The URD was slightly rewritten, the SRD and ADD were combined into a single SDD, and the DDD was eliminated, and replaced by

	Planned	**Actual**
Management documents	18	16
URD	20	12
SRD	18	18
ADD	20	19
DDD	18	20
Code	150	215
Total	**244**	**300**

Table 6.4: Man-hours spent on the collection component before applying the new process.

	Actual
URD	12
SDD	14
Code	80
Maintenance	40
Total	**146**

Table 6.5: Man-hours spent on the collection component after applying the new process.

writing API documentation. The complete source code was rewritten and finally the code could be released to the customer. Table 6.5 shows the man-hours after the project restart.

The number of man-hours spent on the URD after the project restart is skewed. Actually only two man-hours were needed to rewrite the URD. The number is corrected, however, by adding the man-hours spent on meetings and capturing the requirements before the project was restarted. As a result, the tables indicate no improvement. The importance of a URD is recognized by the new process, which explains the same number of man-hours spent on the URD. No management documents need to be written anymore, which saves approximately 16 man-hours.

A significant decrease in man-hours is achieved by replacing the software requirements phase, architectural design phase, and detailed design phase, by the new design phase. Instead of writing three documents, only one document — the SDD [Zwartjes and Geffen, 2004] — is written. The number of man-hours spent on the SDD is distorted as well, because the content is based on the SRD and ADD, written before the restart.

Implementing the design took less time after the project was restarted as well — 80 man-hours compared to 215 man-hours; a decrease of more than 50%. The difference in man-hours can be explained by code reuse, but most of the source code has been completely rewritten. The

exorbitant number of man-hours most probably must be attributed to unskilled developers or their lack of understanding of the design. However, there are no clear grounds to attribute the decrease of implementation cost to the new process.

6.3.2 Case Study II: Internationalization

Initially the administrative application — the main product, wherein the software components are used — was targeted to be sold in the Netherlands only. Intersoft wants to expand its market for its system to foreign countries, but the product has a Dutch user interface. The software must be internationalized. Unfortunately, Delphi's built-in internationalization and localization functionality lacks some important features. For example, it is not possible to use *one* translation of a sentence for all occurrences in the user interface of the application: the same sentence that occurs more than once, has to be translated multiple times. Section 4.6 already described the pitfalls of duplication.

The main problem of internationalizing the administrative application was the fact the application had been developed — in Dutch — for a long time. Common internationalization libraries are based on incorporating multilingual facilities from the start of a project. This was the decisive factor to custom develop the internationalization and localization component. Therefore, the authors were asked to solve this problem by developing software to manage the process of translating the application.

We chose to split the project into a component to internationalize and localize the application and a support application for managing the translations and locale specific information. Table 6.2 includes statistical information about the two projects. The development of the component and application will be elaborated in the following sections.

Upfront, a lot of requirements for both the internationalization component and support application were unstable and not that well known. Because of the small and iterative cycles in the construction phase, new requirements and maintenance and performance problems surfaced immediately in the construction phase. The adaptivity and agility of the customized component development process made it possible to easily anticipate on this changing environment, and satisfy the customer's wishes. In addition, the supporting tools, and especially the automated build and release system that is described in Section 6.2.1, proved their value to support the agility of the process. A new version of the component or application could be made readily available to the customer with the automated build and release system.

Component

The internationalization and localization component is divided into three subcomponents:

- **Internationalization and localization** – Component that provides a mechanism to look up sentences in a specific language and functionality to localize an application.
- **Logging** – Component to log certain messages to a file.
- **User messaging** – Component that provides a mechanism for informing the user.

The component is called docTLUM, which is short for docTranslateLogUserMessaging. The component uses special input files: a translation database for the sentences and translations, and

	Planned	**Actual**
Management documents	16	14
URD	45	25
SRD	25	5
ADD	40	-
DDD	20	-
Code	250	-
Total	**376**	**44**

Table 6.6: Man-hours spent on the internationalization component before the introduction of the new process.

a region database for locale information. The support application that is described in the next section can be used to manage the contents of these files. The big advantage over the translation mechanism provided by Delphi is that each sentence has to be translated *once*: multiple occurrences of the same sentence are substituted by the same translation. This decreases translation costs, because usually a fixed amount of money per translated word has to be paid.

After the documentation of the collections component was finished, the user requirements phase of this component started. While the development team in Sophia was writing the code for project docSet, the documents for the internationalization component could be written. The full ESA standard was applied for the internationalization component. The documents were written with the intention to have the development team in Sophia implement this component as well, after they finished the collections component.

Table 6.6 summarizes the amount of man-hours planned and the man-hours that were actually spent up to the software requirements phase. The development process was interrupted in this phase. The fiasco with the development team in Sophia delivering poor quality source code, and the thoughts about lightening the software engineering process, were the cause to interrupt work on project docTLUM. The results can be found in Chapter 4.

The project was restarted to apply the new process. Table 6.7 shows the man-hours that were spent after applying the new process. The URD was already externally accepted by the customer. To correct the distorted number of hours for the URD, the man-hours spent on interviews and capturing requirements are added to the number of man-hours spent converting the URD [Geffen and Zwartjes, 2003b]. Because this is a more complex project, an SSD was written instead of an SDD. Approximately 40 man-hours have been spent on the SSD [Geffen and Zwartjes, 2003a], which is less than the estimated hours for the SRD and ADD together — 65 man-hours. The estimates for the SRD and ADD of the collections component turned out to be quite accurate. Assuming that the original estimates for this project were accurate as well, the number of man-hours is decreased by approximately 30% in this case.

	Actual
URD	27
SSD	40
Code	240
Maintenance	60
Total	**367**

Table 6.7: Man-hours spent on the internationalization component after project restart and applying the new component development process.

The 30% improvement may not be an improvement if the man-hours saved in this stage of the project are spent in a later stage. For example a lacking design. However, the actual hours that have been spent on coding — 240 man-hours — are close to number of 250 man-hours estimated in the first planning of the project. Secondly, the design was unchanged and proved to be sufficient while it was implemented.

Another 60 man-hours have been spent on maintenance so far. This is likely to increase because the component is being fixed, improved and extended. About 25% of the maintenance hours were spent on bugfixes and the remaining 75% were spent adding new features that were not originally captured in the URD.

For the component, the customized development process was particularly useful because in the construction phase, several new requirements were needed. While the component was implemented, it was tested on a scale, that turned out to be too small. The large number of sentences became unmanageable. To this end, the customer wanted to have status support for sentences, the ability to divide sentences into multiple translation databases, and to extend the information that is logged by the component. The adaptivity of the new process made it possible to solve these problems immediately and quickly.

Support application

An application, called appTranslator, is developed to improve the maintainability of the translation and region (locale) database — the input for the internationalization and localization component. The application mainly consists of two editors: an editor for a translation database and for a region database. Besides these editors, the application contains additional functions to simplify working with the two databases. For example:

- Adding support for new languages to a translation database.
- Import missing sentences from a log file that is generated by the internationalization component (it is possible to enable the component to log all sentences that could not be located in the translation database).
- Merge two translation databases, either to combine two supported languages or to merge translations.

	Actual
URD	30
Prototype	10
SDD	40
Code	250
Maintenance	100
Total	**430**

Table 6.8: Man-hours spent on the support application for the internationalization component.

- Functionality to convert a translation database to a Comma Separated Values (CSV) file and vice versa. This is useful to edit a translation database with an external application.
- Functionality to import sentences from a Structured Query Language (SQL) database or to translate the contents of a SQL database into another language.
- Preview a date, time, currency or number according to the locale information from a region database.

Table 6.8 gives an indication of the hours spent on the support application for the internationalization component. The process described in Chapter 4 was applied to this project too. The exploration phase took 40 hours — 30 hours for writing the URD [Geffen, 2003] and an extra 10 hours for creating the prototype. In case of an application it is worth the effort to prototype the user interface. This may help both the customer and developer to better visualize the application and may lead to defining new or correcting requirements, see Chapter 4.

The exploration phase for this application took more man-hours than for the the collections and internationalization component. Gathering the requirements for this application was less easy. Neither the customer nor the developer had in mind exactly the requirements for the application beforehand. Several meetings and interviews were necessary before all requirements had surfaced. An SDD [Geffen, 2004] was written for this project and the design phase was finished in approximately 40 hours.

Two factors caused an increase in the number of man-hours spent in the construction phase. The grid component that was used to visualize the translation database was found to lack some features at some point. It was decided to substitute the grid component with a more advanced proprietary third-party grid, that implements the lacking features. It took approximately 40 additional man-hours to incorporate the advanced grid. However, an analysis showed that implementing the lacking features in the original grid would have taken even more time. Secondly, the project was the lead developer's first independently developed application. He already implemented parts of the components, some additional time was needed to familiarize with the details of implementing a standalone application. The high number of man-hours in the maintenance phase are caused by new major requirements, that took a significant amount of man-hours to implement.

	Actual
URD	10
SDD	15
Code	80
Maintenance	40
Total	**145**

Table 6.9: Man-hours spent on the application security component.

Application appTranslator was tested on a small scale: the application itself served as its own test case. This led to performance issues when the translation database for Interclean was created. In comparison with the translation database of appTranslator, Interclean's translation database contains about fifteen times more sentences and supports six more languages. Maintenance of the translation database turned out to be difficult because of the size of the translation database. Because appTranslator was tested by the Interclean developers immediately when there was a first working version of the application available, the performance problems were detected early in the development of appTranslator. This allowed us to slightly redesign and refactor the code to improve performance for large translation databases, and use the new design to implement the remaining requirements.

6.3.3 Case Study III: Security

To include experiences of two other employees with the process from Chapter 4, a third case study was conducted, which is about security. To protect end-users from violating their licenses, rules for user rights are embedded into an application. To enhance that part of the application, and to make user rights and licensing more maintainable, several components are build to abstract and encapsulate security and licensing of an application. Two components will be highlighted in this section: vizLicense[2] and docEncrypt. Project vizLicense implements licensing and application security, and docEncrypt implements encryption and decryption of data. The last row in Table 6.2 lists project vizLicense. No information is included in that table about docEncrypt as the source code is not yet finished at the time of writing.

Licensing and user rights

To implement security in an application, the visual user interface components need support for application security. The vizLicense component is an extra layer, that can be used in conjunction with a previously developed component for application security. It adds the necessary support to visual items of the user interface. The man-hours spent in the project can be found in table 6.9. The developer responsible for this project, Wouter Bijlsma, also developed the previous layer —

[2]Components with a visual interface are prefixed with *viz* instead of *doc*.

implementing the core application security concepts. He tried out the process from Chapter 4, on both projects, and we interviewed Bijlsma to find out what he thinks of the process.

> Before I started with the design of the docLicense and vizLicense components I already had some experience applying a more rigorous design process to the development of medium-sized software projects. This experience resulted from an internship and two software engineering projects, one in the role of a project member and the other in the role of project manager. This experience serves very well for comparison with the agile development process applied to vizLicense and, in less degree, docLicense[3].
>
> The docLicense component was the first component I designed [in my new job], and I started off using a development model that was still quite rigorous. [The process from Chapter 4 was under development at the time. Bijlsma used an intermediate version for his project.] This turned out not to be completely overdone because the component is quite complex and has a lot of independent user requirements that can be hard to explain to other developers, but during the design process I learned that a few things could be done more efficiently. For docLicense I wrote a full-fledged SSD (as opposed to an SDD), and I would probably do the same now because of the extent and complexity of the component. However, some aspects of the original SSD serve no critical purpose, specifically the class interface and property tables. The API docs that are generated from the code using the API documentation tool are far more useful, contain a lot more information that is automatically extracted from the structure of the code, and are much more maintainable than the SSD interface descriptions. Best of all, the generated API docs can actually be used during the development of new software that is using the components, because they can be viewed and navigated on-line. Paper documents are not convenient when you are programming.
>
> After the docLicense component was finished I started the vizLicense component. For this component the agile development model was used exclusively. This resulted in a very swift completion of the URD and SDD, without sacrificing their usability. The vizLicense component is quite simple in terms of its functionality and design so everything that does not contribute to the final code design or the description of its functionality would be overdoing. The URD and SDD both took under 15 man hours, where the docLicense URD and SSD took 25 and 32, respectively. In comparison, writing an URD, SSD, ADD and DDD, as prescribed by the (rigorous) ESA software engineering standard used for previous projects I took part in took around 250 man hours for documentation alone. I do not have the impression that the extra time spent adds to the quality of the final software product. In fact, it appears to me that the extra time available for implementation issues only improves the quality of the code.
>
> Summarizing, my experience with rigorous and agile design processes is that the added overhead of documenting everything to the finest detail does not add to the quality of the final software product. This does not mean that documenting the user requirements and design of your software is useless. It is indispensable. However, by using the right tools that can substitute design documentation that otherwise should

[3]The name of the project to develop the component that implements the core functionality for application security

	Actual
URD	25
SDD	32
Code	39
Maintenance	0
Total	**96**

Table 6.10: Man-hours spent on the security component.

have been written by hand, and by stripping out everything that does not add to the understanding of the component, its design, or the functionality it should provide, a lot of time can be saved. The saved time can then be reallocated to improve the quality of the final software component.

Encryption and decryption

To protect sensitive information in an application — such as passwords, amounts of money, and so on — from getting publicly available, an encryption and decryption component is needed. The project to build that component is assigned to a newly hired employee, as his first project. The man-hours he spent on the URD and SDD are depicted in Table 6.10. The component is currently being implemented, hence the number of man-hours spent to code in Table 6.10 is not final.

The number of hours spent on the documents is slightly higher than in the previous discussed projects. Most probably this increase can be attributed to the fact that the developer is new, and that it is his first project. We also interviewed the responsible developer, Rick van Bijnen, to find out his experiences working with the process for the first time. A summary:

> This is my first project, and therefore my only experience with the [component] development methods. Because [the project] is not even finished, I haven't been through the entire procedure. My experience is therefore limited to the exploration phase, design phase and construction phase.
>
> Exploration phase: This phase is completed. Basically it boils down to constructing the URD. I find the URD a very useful document, it clearly outlines the goals of the project. It also forced me to do proper research on encryption and its possibilities before starting on the design and implementation work. The other team members did not do this research, but I think this should be an obligation. This would avoid discussions about requirements that turn out to be a misunderstanding of the material. Also, proper background knowledge is essential for the reviewing phase of the URD. This part should be taken more seriously in the future.
>
> Design phase: The design phase is completed. I [made the choice to create] an SDD for this project. The [document] makes sure that you think thoroughly about the design of your software instead of just starting off programming. On the other hand,

> sometimes you have to decide about issues that you should better postpone until a later stage, but I think that this is a minor disadvantage compared to the benefits of the well thought out design that is a result of the SDD. I do not feel that I am over-documenting.
>
> Implementation phase: This phase is currently in progress. I cannot comment too much about this. Coding standards are a good thing to have. The design is already recorded in the SDD, which simplifies the programming significantly and really speeds up the implementation process.

The point van Bijnen makes about the exploration phase can be debated. Research certainly is part of the exploration phase, and all team members should dig into the material. However, in a review, the authors of the document should be able to explain and convince the reviewers about requirements when necessary. At least the author should be able to point the reviewers to literature explaining the issues that were not understood.

6.4 Conclusion

This chapter analyzed the practical work that was undertaken to custom develop two tools and the effect of these developments. In addition, the results of testing out the proposed development process in several projects were presented.

The study of the automated build system showed that the tool decreased the time to build and deploy a component. An investment of man-hours however was needed in advance, to develop the build system. The code documenting tool had no measurable effects on the development process. However, a lot of positive reactions have been received from developers that are using the generated API documentation.

The proposed development process was applied successfully to the six projects described in this chapter. No major problems were encountered in any of the phases and the developers reacted positively. The process scales to components as well as applications. In the cases where comparable data was available, the new process decreased the number of hours required for writing documentation and the number of hours for the implementation remained the same.

The adoption on the newly formulated, customized process, has led to improvements of the entire development process. The development of the internationalization component and support application, for example, have benefited from the new process, because the requirements were unstable and the design needed to be changed. The adaptivity of the process made it possible to go back to previous phases more easily and fix the problems as soon as possible. The tools that support the process have a significant impact on the adaptability of the process as well. An automated build and release system, for example, decreases deployment time, which is particularly useful in a process with small, iterative cycles. It is the combination of both process and tools, that improve the productivity and effectiveness of the developers in the component development context. The benefit of tools is maximized with a compatible process, and the benefit of a process can be maximized with compatible tools.

Chapter 7

Conclusion

The research questions in Section 1.1 have been answered indirectly throughout the dissertation. In this chapter, these questions are answered directly by a short summary of what has been concluded in the previous chapters. In addition, the quality and origin of the answers and relevant future work is discussed.

1. What is the current scope of software engineering methodology, and more specific, what are the main trends or directions in the area?

Historically, software engineering is associated with a heavyweight character. To overcome problems with heavyweight methods, a new trend in software engineering — called lightweight or agile — has emerged. Additionally, another related direction has evolved — called the open source movement — based on a high level of collaboration around publicly accessible source code. As a result, three trends were identified, all having their own unique characteristics: (1) heavyweight software engineering, (2) agile software engineering, and (3) open source software engineering. Chapter 2 contains a summary of the relevant literature on this topic.

2. What are the differences between the directions of software engineering methodologies, and can software engineering methods be classified?

The weight is the key difference between heavyweight and agile. Tables 2.1 and 2.2 list specific characteristics and differences between agile and heavyweight software engineering. Open source software engineering resembles agile software engineering to a great extent. Cockburn [Abrahamsson *et al.*, 2002], however, argues that open source software development differs from the agile development model in philosophical, economical and team structural aspects.

It is difficult to classify a software engineering methodology to be heavyweight because no precise definition of heavyweight software engineering exists. However, the Agile Manifesto [Beck *et al.*, 2001] can be used to classify a software engineering method as agile, and the OSD [Open Source Initiative, 2004] can be used for a project to be certified open source. Chapter 2 discusses the differences between the directions of software engineering methodologies, as well as classifying a method.

3. What are the main considerations that influence the choice of a software engineering methodology for a project?

Section 2.7 addresses the answer to this question. The number of people in a project, the criticality of a project, the available budget, and team structural aspects are the most important factors that must be considered when a methodology for a project is chosen. Not all of the factors have to be fixed in advance for a given project. Fixed factors will influence the choice for the variable ones. In business, the budget of a project is often the most important factor.

4. What software engineering methodology is best suited for the development of software components?

The methodology selection criteria, from Section 2.7 were applied to the component development environment, described in Section 4.9. The number of people involved is small, the criticality low, the available budget is minimal and the team is co-located in the same office. As a result, the agile methodology was concluded to be the most suitable approach. Whenever the parameters — such as criticality and scale — of the component development process change, the choice of methodology should be reconfirmed.

5. What software engineering method is best suited for the development of software components?

It was decided to formulate a custom process, described in Chapter 4, for the development of the components, based on previous experience with the already existing ESA lite software engineering standard. The ESA lite standard was chosen originally — before the research described in this dissertation was started — but turned out to be too heavy. This was an important factor to start this research.

Empirical data from the case studies in Chapter 6 confirm that the newly formulated component development process is more efficient. The component development process can be classified as agile according to the Agile Manifesto and also by comparing the component development process to the characteristics described in the tables defined in the answer to the second research question. This analysis can be found in Section 4.9. The case studies in Chapter 6 involve small projects. Whether the process scales to medium sized or large projects is to be investigated further.

6. What impact do tools have on a project or, more general, what is their impact on a development process?

Tools can decrease the development effort — and thus the cost — of a project. The development effort is decreased by assisting in managing mistakes in three ways: (1) by preventing errors, (2) by detecting and locating errors, and (3) by fixing errors. Furthermore, they have a significant impact in a software engineering process because they can be a valuable resource to assist in adhering to a certain software engineering method. Moreover, tools can help to define the software engineering process — with the open source development process as an example. The importance of tools is discussed in Chapter 3.

7. How to choose the tools for a development process, or more specific for a project?

Usually, tools are mainly chosen based on experience and intuition, which can be imprecise and misleading. A model to assist in choosing a set of tools is introduced in Chapter 5, including examples that illustrate the usefulness of the model. The model makes explicit the practical requirements for the tools that are needed in a process or project, as well as the properties of a collection of tools. As a result, a decision based on the model is made transparent. Future research may focus on a software tool to manage tool matrices, as well as assist in choosing a set of tools based on a tool matrix.

An important property of a tool, especially in business, is its cost — not only its price, but, for example, also the cost of training. The cost of a tool can be made explicit in the tool matrix by including it as one or more properties.

8. Which tools are commonly used to enhance the development process?

Based on our experience, a study of literature on this topic, and an analysis of several projects, there are three tools that are commonly used to improve the development process: (1) a version control system, (2) a bug and issue tracking tool, and (3) a build system. However, in another context, other tools may also improve the development process, for example, testing tools may be valuable. The contexts that are used to answer this question are detailed in Section 3.4.

The amount of discipline that is necessary to introduce and use a tool, is a major influence on a tool's value in the development process. This amount of discipline is different for each tool. Whether a tool is successful and valuable in a specific context, may be explained by the discipline factor. This can be an interesting research topic, which is not addressed in this dissertation. The discipline factor is closely related to the cost properties of a tool.

9. Which tools are important to support the process for the development of software components?

The model, from the answer to question 7, was applied to choose a set of tools for the newly formulated component development process, in Section 5.4.1. Although the model was applied retrospectively, it proved that the current set of tools is sufficient and justifies the choice to develop two custom tools. For the development of a tool matrix management and decision support application, the data from the case studies in Sections 5.4.1 and 5.4.2 may be useful.

The model should be applied again in the future, because new tools may be available or needs may change. Additionally, if team members leave or new team members join, tool preference may be different, considering, for example, the discipline factor.

10. What is the relation between people and a software engineering process?

The people involved is the most important factor in a process or project. From Chapter 2, it can be concluded that a lightweight process requires creative and responsive people. A greater sense of freedom in a lightweight process, stimulates developer creativity. A heavyweight process concentrates on providing something to hold on to. A team with a majority of less creative people, will benefit more from a heavyweight process. The major difference between agile and heavyweight

processes, in people, is that an agile process is molded to the specific team, and in a heavyweight process, the team is molded to the process.

List of Abbreviations

ACPI Advanced Configuration and Power Interface
ADD Architectural Design Document
API Application Programming Interface
ASD Adaptive Software Development

BSD Berkeley Software Distribution
BSSC Board for Software Standardization and Control

CASE Computer Aided Software Engineering
CMM Capability Maturity Model
CP Construction Phase
CSD Coding Standards Document
CSV Comma Separated Values
CVS Concurrent Versions System

DDD Detailed Design Document
DP Design Phase
DRY Don't Repeat Yourself
DSDM Dynamic Systems Development Methodology
DSS Document Status Sheet

ECSS European Cooperation for Space Standardization
EP Exploration Phase
ESA European Space Agency

FDD Feature Driven Development
FIFO First In First Out
FSBN Fonds Studiepunten Buiten Nederland

GDE GTK Development Environment
GENESIS Generalized Environment for Process Management In Cooperative Software Engineering

GIMP General Image Manipulation Program
GNU GNU's Not Unix
GPL General Public License
GTK The GIMP Toolkit
GUI Graphical User Interface

HTML Hypertext Markup Language

IDE Integrated Development Environment
IEEE Institute of Electrical and Electronic Engineers

KDE K Desktop Environment
KIVI Koninklijk Instituut Van Ingenieurs

LGPL Library GPL
LIFO Last In First Out

MP Maintenance Phase
MTBF Mean Time Between Failure
MPL Mozilla Public License

OOAD Object Oriented Analysis and Design
OOP Object Oriented Programming
OPHELIA Open Platform and Methodologies for Development Tools Integration in a Distributed Environment
OSD Open Source Definition
OSI Open Source Initiative

PHP PHP Hypertext Preprocessor
PSE Programming Support Environment
PSEE Process-centered Software Engineering Environment
PSS Procedures, Standards and Specifications

QPL Q Public License

RUP Rational Unified Process

SAIEE South African Institute of Electrical Engineers
SCMP Software Configuration Management Plan
SD Software Design
SDD Software Design Document
SEE Software Engineering Environment

SEI Software Engineering Institute
SLOC Source Lines Of Code
SPMP Software Project Management Plan
SQAP Software Quality Assurance Plan
SQL Structured Query Language
SRD Software Requirements Document
SSD Software Specification Document
SUM Software User Manual
SVVP Software Verification and Validation Plan

TU/e Technische Universiteit Eindhoven

UML Unified Modeling Language
URD User Requirements Document

WYSIAYG What You See Is All You Get
WYSIWYG What You See Is What You Get

XML Extensible Markup Language
XP Extreme Programming

Bibliography

[Abrahamsson *et al.*, 2002] Pekka Abrahamsson, Outi Slao, Jussi Ronkainen, and Juhani Warsta. Agile software development methods: Reviews and analysis. *ESPOO 2002*, 2002.

[Auer and Miller, 2003] Ken Auer and Roy Miller. *Extreme Programming Applied: Playing to Win*. Addison-Wesley, 2003.

[Beck *et al.*, 2001] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. The agile manifesto. [Online]. Available: `http://www.agilemanifesto.org`, 2001.

[Beck, 1999] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[Boldyreff *et al.*, 2003] Cornelia Boldyreff, Mike Smith, Dawid Weiss, David Nutter, Pauline Wilcox, Stephen Rank, and Rick Dewar. Environments to support collaborative software engineering. In *Cooperative Methods and Tools for Distributed Software Processes, 2nd Workshop on Cooperative Supports for Distributed Software Engineering Processes*. FrancoAngeli, 2003.

[Booch, 1991] Grady Booch. *Object-Oriented Analysis And Design With Application*. Benjamin-Cummings, 1991.

[Brooks Jr., 1987] Frederick P. Brooks Jr. No silver bullet - essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, 1987.

[Brooks Jr., 1995] Frederick P. Brooks Jr. *The Mythical Man Month: Essays on Software Engineering*. Addison-Wesley, 1995.

[Browne, 1998] Cristopher B. Browne. Linux and decentralised development. *First Monday*, 3(3), March 1998.

[Coad and Yourdon, 1991] Peter Coad and Edward Yourdon. *Object-Oriented Design*. Prentice Hall, 1991.

[Cockburn, 2000] Alistair Cockburn. Selecting a project 's methodology. *IEEE Software*, 17(4), 2000.

[Coleman *et al.*, 1994] Derek Coleman, Patrick Arnold, Stephanie Bodoff, Chris Dolin, Helena Gilchrist, Fiona Hayes, and Paul Jeremaes. *Object-Oriented Development: The Fusion Method*. Prentice Hall, 1994.

[Collabnet, Inc., 2004] Collabnet, Inc. Tigris.org: Open source software engineering. [Online]. Available: `http://tigris.org/`, 2004.

[DiBona *et al.*, 1999] Chris DiBona, Sam Ockman, and Mark Stone. Introduction. In Chris DiBona, Sam Ockman, and Mark Stone, editors, *Open Sources: Voices from the Open Source Revolution*. O'Reilly and Associates, Cambridge, Massachusetts, 1999.

[Engels *et al.*, 2001] Gregor Engels, Wilhelm Schäfer, Robert Balzer, and Volker Gruhn. Process-centred software engineering environments: academic and industrial perspectives. In *Proceedings of the 23rd international conference on Software engineering*, pages 671–673. IEEE Computer Society, 2001.

[European Space Agency, 1991] European Space Agency. *ESA PSS-05-0 ISSUE 2: ESA Software Engineering Standards*. European Space Agency, 1991.

[European Space Agency, 1995a] European Space Agency. *ESA PSS-05-01 ISSUE 1 REVISION 1: Guide to the Software Engineering Standards*. European Space Agency, 1995.

[European Space Agency, 1995b] European Space Agency. *ESA PSS-05-02 ISSUE 1 REVISION 1: Guide to the User Requirements Definition Phase*. European Space Agency, 1995.

[European Space Agency, 1995c] European Space Agency. *ESA PSS-05-03 ISSUE 1 REVISION 1: Guide to the Software Requirements Definition Phase*. European Space Agency, 1995.

[European Space Agency, 1995d] European Space Agency. *ESA PSS-05-04 ISSUE 1 REVISION 1: Guide to the Software Architectural Design Phase*. European Space Agency, 1995.

[European Space Agency, 1995e] European Space Agency. *ESA PSS-05-05 ISSUE 1 REVISION 1: Guide to the Software Detailed Design and Production Phase*. European Space Agency, 1995.

[European Space Agency, 1995f] European Space Agency. *ESA PSS-05-06 ISSUE 1 REVISION 1: Guide to the Software Transfer Phase*. European Space Agency, 1995.

[European Space Agency, 1995g] European Space Agency. *ESA PSS-05-07 ISSUE 1 REVISION 1: Guide to the Software Operations and Maintenance Phase*. European Space Agency, 1995.

[European Space Agency, 1995h] European Space Agency. *ESA PSS-05-08 ISSUE 1 REVISION 1: Guide to Software Project Management*. European Space Agency, 1995.

[European Space Agency, 1995i] European Space Agency. *ESA PSS-05-09 ISSUE 1 REVISION 1: Guide to Software Configuration Management*. European Space Agency, 1995.

[European Space Agency, 1995j] European Space Agency. *ESA PSS-05-10 ISSUE 1 REVISION 1: Guide to Software Verification and Validation Phase*. European Space Agency, 1995.

[European Space Agency, 1995k] European Space Agency. *ESA PSS-05-11 ISSUE 1 REVISION 1: Guide to Software Quality Assurance Phase*. European Space Agency, 1995.

[European Space Agency, 1996] European Space Agency. *ESA BSSC(96)2 ISSUE 1: Guide to Applying the ESA Software Engineering Standards to Small Software Projects*. European Space Agency, 1996.

[European Space Agency, 2004] European Space Agency. Collaboration website of the ECSS. [Online]. Available: `http://www.ecss.nl`, 2004.

[Feller and Fitzgerald, 2000] Joseph Feller and Brian Fitzgerald. A framework analysis of the open source software development paradigm. In *Proceedings of the twenty first international conference on Information systems*, pages 58–69. Association for Information Systems, 2000.

[Free Software Foundation, 2004a] Free Software Foundation. Free software foundation. [Online]. Available: `http://www.gnu.org/fsf/fsf.html`, 2004.

[Free Software Foundation, 2004b] Free Software Foundation. Gnu general public license. [Online]. Available: `http://www.gnu.org/copyleft/gpl.html`, 2004.

[Free Software Foundation, 2004c] Free Software Foundation. Gnu operating system. [Online]. Available: `http://www.gnu.org/`, 2004.

[Gamma *et al.*, 1995] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Geffen and Zwartjes, 2003a] Joost van Geffen and Gertjan Zwartjes. *Software Specification Document: docTLUM*. Intersoft Software Research, 2003.

[Geffen and Zwartjes, 2003b] Joost van Geffen and Gertjan Zwartjes. *User Requirements Document: docTLUM*. Intersoft Software Research, 2003.

[Geffen, 2003] Joost van Geffen. *User Requirements Document: appTranslator*. Intersoft Software Research, 2003.

[Geffen, 2004] Joost van Geffen. *Software Design Document: appTranslator*. Intersoft Software Research, 2004.

[Goerzen, 2000] John Goerzen. *Linux Programming Bible*. IDG books Worldwide, 2000.

[Highsmith and Cockburn, 2001a] Jim Highsmith and Alistair Cockburn. Agile software development: The business of innovation. *IEEE Computer Society*, 34:120–122, September 2001.

[Highsmith and Cockburn, 2001b] Jim Highsmith and Alistair Cockburn. Agile software development: The people factor. *IEEE Computer Society*, 34:131–133, September 2001.

[Highsmith *et al.*, 2001] Jim Highsmith, Jeff Sutherland, Robert L. Glass, Larissa T. Moss, Philippe Kruchten, Larry Wagner, and Lou Russel. The great methodologies debate: Part 1. *Cutter IT Journal*, 14, December 2001.

[Highsmith *et al.*, 2002] Jim Highsmith, Alistair Cockburn, Stephen J. Mellor, Ivar Jacobson, Brian Henderson-Sellers, and Matt Simons. The great methodologies debate: Part 2. *Cutter IT Journal*, 15, January 2002.

[Hunt and Thomas, 1999] Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley, 1999.

[Iivari, 1996] Juhani Iivari. Why are case tools not used? *Communications of the ACM*, 30(10):94–103, October 1996.

[Jackson, 1995] Michael Jackson. *Software requirements & specifications: a lexicon of practise, principles and prejudices*. ACM Press/Addison-Wesley Publishing Co., 1995.

[Jankowski, 1997] David J. Jankowski. Computer-aided systems engineering methodology support and its effect on the output of structured analysis. *Empirical Software Engineering: an International Journal*, (1):11–38, 1997.

[Jeffries *et al.*, 2000] Ron Jeffries, Ann Anderson, and Chet Hendrikson. *Extreme Programming Installed*. Addison-Wesley, 2000.

[Jones *et al.*, 1997] M. Jones, C. Mazza, U.K. Mortensen, and A. Scheffer. Twenty years of software engineering standardisation in ESA. [Online]. Available: `http://esapub.esrin.esa.it/bulletin/bullet90/b90jones.htm`, 1997.

[Kernighan and Mashey, 1979] Brian W. Kernighan and John R. Mashey. The unix programming environment. *Software Practise and Experience*, 9(1):1–15, 1979. Also in *IEEE Computer*, Vol. 14 (4), April 1981.

[Khan, 2004] Ali Khan. A tale of two methodologies for web development: Heavyweight versus agile. *Tenth Australian World Wide Web Conference (AusWeb)*, 2004.

[Kitchenham *et al.*, 1997] Barbara A. Kitchenham, Stephen G. Linkman, and D. Law. Desmet: a methodology for evaluating software engineering methods and tools. *Computing and Control Engineering Journal*, pages 120–126, 1997.

[Koch and Schneider, 2000] Stefan Koch and Georg Schneider. Results from software engineering research into open source development projects using public data. Diskussionspapiere zum Tätigkeitsfeld Informationsverarbeitung und Informationswirtschaft, H.R. Hansen und W.H. Janko (Hrsg.), Nr. 22, Wirtschaftsuniversität Wien, 2000.

[Koch, 2004] Stefan Koch. Agile principles and open source software development: A theoretical and empirical discussion. *Lecture Notes in Computer Science*, 3092:85–93, 2004.

[LeBlanc and Korn, 1994] Louis A. LeBlanc and Willard M. Korn. A phased approach to the evaluation and selection of case tools. *Information and Software Technology*, 36(5):267–273, 1994.

[Maccari and Riva, 2000] Alessandro Maccari and Claudio Riva. Empirical evaluation of case tools usage at nokia. *Empirical Software Engineering*, 5(3):287–299, 2000.

[Moody, 2002] Glyn Moody. *Rebel Code: Linux and the Open Source Revolution*. Persues Publishing, 2002.

[Mozilla, 2004] Mozilla. Mozilla.org - home of mozilla, firefox, thunderbird, and camino. [Online]. Available: `http://www.mozilla.org/`, 2004.

[Naur and Randell, 7 11 October 1968] P. Naur and B. Randell, editors. *Software Engineering: Report of a conference sponsored by the NATO Science Committee*. Brussels, Scientific Affairs Division, NATO, 7-11 October, 1968.

[Object Management Group, 2004] Object Management Group. Uml resource page. [Online]. Available: `http://www.uml.org/`, 2004.

[Open Source Initiative, 2004] Open Source Initiative. The open source definition. [Online]. Available: `http://www.opensource.org/docs/definition_plain.html`, 2004.

[OSDN, 2004a] OSDN. Freshmeat.net. [Online]. Available: `http://www.freshmeat.net/`, 2004.

[OSDN, 2004b] OSDN. Sourceforge.net. [Online]. Available: `http://www.sourceforge.net/`, 2004.

[Ossher *et al.*, 2000] Harold Ossher, William Harrison, and Peri Tarr. Software engineering tools and environments: a roadmap. In *Proceedings of the conference on The future of Software engineering*, pages 261–277. ACM Press, 2000.

[Osterweil, 1987] Leon Osterweil. Software processes are software too. In *Proceedings of the 9th international conference on Software Engineering*, pages 2–13. IEEE Computer Society Press, 1987.

[Oxford University, 6th edition 2002] Oxford University. *Oxford Advanced Learner's Dictionary*. Oxford University Press, 6th edition 2002.

[Perens, 1999] Bruce Perens. The open source definition. In Chris DiBona, Sam Ockman, and Mark Stone, editors, *Open Sources: Voices from the Open Source Revolution*. O'Reilly and Associates, Cambridge, Massachusetts, 1999.

[Plessis, 1993] Annette L. du Plessis. A method for case tool evaluation. *Information and Management*, 25(2):93–102, 1993.

[Randell and Buxton, 27 31 October 1969] B. Randell and J.N. Buxton, editors. *Software Engineering Techniques: Report of a conference sponsored by the NATO Science Committee*. Brussels, Scientific Affairs Division, NATO, 27-31 October, 1969.

[Raymond, 2001] Eric Steven Raymond. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly and Associates, 2001.

[Reis and de Mattos Fortes, 2002] Christian Robotto Reis and Renata Pontin de Mattos Fortes. An overview of the software engineering process and tools in the mozilla project. In *Proceedings of the Open Source Software Development Workshop*, pages 155–175, February 2002.

[Robbins, 2003] Jason E. Robbins. Adopting open source software engineering (osse) practises by adopting osse tools. [Online]. Available: `http://www.ics.uci.edu/~jrobbins/papers/robbins-msotb.pdf`, 2003.

[Rumbaugh *et al.*, 1991] James R Rumbaugh, Michael R. Blaha, William Lorensen, Frederick Eddy, and William Premerlani. *Object-oriented Modelling and Design*. Prentice Hall, 1991.

[Sharma and Rai, 2000] Srinarayan Sharma and Arun Rai. Case deployment in is organisations. *Communications of the ACM*, 43(1):80–88, 2000.

[Shlaer and Mellor, 1992] Sally Shlaer and Stephen J. Mellor. *Object Lifecycles: Modelling the World in State*. Prentice Hall, 1992.

[Shlaer, 1988] Sally Shlaer. *Object-Oriented Systems Analysis: Modelling the World in Data*. Prentice Hall, 1988.

[Stobart *et al.*, 1993] S.C. Stobart, Anton J. van Reeken, and Graham C. Low. An empirical evaluation of the use of case tools. In *Proceedings of the sixth international workshop on Computer-aided software engineering*, pages 81–87. IEEE Computer Society, 1993.

[Szyperski *et al.*, 2002] Clemens Szyperski, Dominik Gruntz, and Stephan Murer. *Component Software*. Addison Wesley, 2002.

[Theunissen *et al.*, 2003] Morkel Theunissen, Derrick Kourie, and Bruce Watson. Standards and agile software development. In *Proceedings of SAICSIT 2003: Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists*, pages 178–188, Fourways, South Africa, September 2003. SAICSIT, ACM.

[Torvalds and Diamond, 2002] Linus Torvalds and David Diamond. *Just for Fun: The Story of an Accidental Revolutionary*. HarperBusiness, 2002.

[Trolltech AS, 2004] Trolltech AS. Q public license. [Online]. Available: `http://www.trolltech.com/licenses/qpl-annotated.html`, 2004.

[Vixie, 1999] Paul Vixie. Software engineering. In Chris DiBona, Sam Ockman, and Mark Stone, editors, *Open Sources: Voices from the Open Source Revolution*. O'Reilly and Associates, Cambridge, Massachusetts, 1999.

[Zwartjes and Geffen, 2004] Gertjan Zwartjes and Joost van Geffen. *Software Design Document: docSet*. Intersoft Software Research, 2004.

[Zwartjes, 2002] Gertjan Zwartjes. *Coding Standards Document*. Intersoft Software Research, 2002.

[Zwartjes, 2003] Gertjan Zwartjes. *User Requirements Document: appDelphiDoc*. Intersoft Software Research, 2003.

[Zwartjes, 2004] Gertjan Zwartjes. *Software Design Document: appDelpiDoc*. Intersoft Software Research, 2004.