JYVÄSKYLÄ STUDIES IN COMPUTING 16

Marko Forsell

Improving Component Reuse in Software Development

Esitetään Jyväskylän yliopiston informaatioteknologian tiedekunnan suostumuksella julkisesti tarkastettavaksi Kokkolan yhteislyseon lukion auditoriossa maaliskuun 8. päivänä 2002 kello 12.

Academic dissertation to be publicly discussed, by permission of the Faculty of Information Technology of the University of Jyväskylä, in the Auditorium of Kokkola Yhteislyseo Senior Secondary School, on March, 8, 2002 at 12 o'clock noon.

UNIVERSITY OF JYVÄSKYLÄ

JYVÄSKYLÄ 2002

Improving Component Reuse in Software Development

JYVÄSKYLÄ STUDIES IN COMPUTING 16

Marko Forsell

Improving Component Reuse in Software Development

UNIVERSITY OF JYVÄSKYLÄ

JYVÄSKYLÄ 2002

Editors
Seppo Puuronen
Department of Computer Science and Information Systems, University of Jyväskylä
Marja-Leena Tynkkynen and Pekka Olsbo
Publishing Unit, University Library of Jyväskylä

URN:ISBN:9513912175
ISBN 951-39-1217-5 (PDF)

ISBN 951-39-1161-6 (nid.)
ISSN 1456-5390

Copyright © 2002, by University of Jyväskylä

ABSTRACT

Forsell, Marko
Improving Component Reuse in Software Development / Marko Forsell
Jyväskylä: University of Jyväskylä, 2002, 167 pages
(Jyväskylä Studies in Computing
ISSN 1456-5390;16)
ISBN 951-39-1217-5
Finnish summary
Diss.

This study concerns reuse in the software development process. The focus is in the reuse of components when creating new software. The aim is to improve current software processes to adapt them to the purposes of reuse. The specific research questions are: 1) What are the specific limitations for reuse in the current component-based software development methods? 2) How should reusable components be created and used in software development? 3) How should the components that are created be documented in order to make them reusable in other development projects? The study follows the reflective systems development approach. First, current component-based software development methods are evaluated and examined in the light of reuse process. Also, the software process currently used in organizations is examined. This improves our understanding of the current situation. Here it is claimed that successful and systematic reuse-orientation requires use of the domain analysis method and a systematic way of documenting components. Next, some improvements are designed to address the problems identified. Here the hierarchical domain analysis method and a model to document reusable components are presented. Finally, presented solutions are tried out in practice.

Keywords: Software reuse, reuse process, component-based development, domain analysis, component documentation

ACM Computing Review Categories

D.2.1	Software Engineering: Requirements/Specifications: Methodologies, Domain analysis, Component documentation
D.2.10	Software Engineering: Design Methodologies, Domain analysis, Component documentation
D.2.13	Software Engineering: Reusable Software Domain engineering, Reuse models, Domain analysis, Component documentation
D2.m	Software Engineering: Miscellaneous Reusable software

Author's Address

Marko Forsell
University of Jyväskylä
Chydenius Institute
P.O. Box 567
FIN-67701 KOKKOLA
Finland
e-mail: marko.forsell@chydenius.fi
Fax: +358 6 8294 202

Supervisors

Jarmo Ahonen
Information Technology Research Institute
University of Jyväskylä, Finland

Markku Sakkinen
Department of Computer Science and Information Systems
University of Jyväskylä, Finland

Reviewers

Cornelia Boldyreff
Department of Computer Science
University of Durham, United Kingdom

Eila Niemelä
Technical Research Centre of Finland, Oulu, Finland

Opponent

Jan Bosch
Department of Computing Science
University of Groningen, Netherlands

ACKNOWLEDGMENTS

First of all, I want to thank Anni and Heidi. Anni, you were perhaps one of the main reasons I went back to the university. Without that this thesis would have never be written. Of course, I want to acknowledge the role played by the latest arrival to our family, Maija. I want to thank all of you Anni, Maija, and Heidi.

I want to thank my parents who have supported me during the years. Also, I'd like to extend my thanks to my grandparents who continously expressed the importance of education. Thank you: dad, mom, tuta and Helena.

The first time I thought about post-graduate studies were during the ECOOP'97 conference. Markku Sakkinen asked for volunteers in an e-mail message for that conference. Because I had earlier decided to study that summer I replied to that message and offered my help. During the conference I happened to speak with Jarmo Ahonen about post-graduate studies. The main message in that discussion was that post-graduate studies do not require "super-human" intelligence as much as stamina. So, two years later I found myself working in the IT Research Institute, Jarmo Ahonen as its director. It was at that point of time I started to work on this thesis. I asked Markku Sakkinen and Jarmo Ahonen to be my advisors and luckily they accepted this request. I want to thank both of you Jarmo and Markku.

During my studies I co-operated with various people in our university. I am especially in dept to Veikko Halttunen and Tero Päivärinta. In the PISKO project I co-worked with number of people. I want to thank all of you Ari Häkkinen, Esko Hakulinen, Tuukka Junttila, Marko Hollanti, and Sanna-Kaisa Taskinen.

This study was done in PISKO project which was funded by Tekes and five companies participating in that project: Aplicom Oy[1], Honeywell Oy[2], TietoEnator Corp.[3], Yomi Solution Ltd.[4] (formely Relatech Ltd.), and Republica Corp.[5], who have kindly given a written permission to mention them by name. Without these companies and Tekes this thesis would not have been accomplished.

[1] www.aplicom.com
[2] www.honeywell.com
[3] www.tietoenator.com
[4] www.yomi.com/solution/
[5] www.republica.fi

CONTENTS

1 INTRODUCTION AND BACKGROUND

1.1 Introduction

1.1.1 Background and Motivation

In the modern society software has become a significant part of our everyday life. New application areas for software are continously discovered and we can find software in our phones, cars, and television sets. Further, we entrust our lives to software in the form of various applications used in hospital environments. All this software must be developed, and after that it must be enhanced and maintained. Rising demands for new software, software updates, and software maintenance require human work. Currently software companies must develop quality software more efficiently, and in a more cost effective way.

If the same production considerations were applicable to software as are to the conventional industry products we would not have problems since production (i.e., producing a new copy) of software is very easy and inexpensive. If conventional industry wants to improve its production processes, they concentrate on the production phase. In the software industry the bottleneck is not the production but the development. The main cost of software production comes from developing it. Every time a program must be altered, ported to another platform, enhanced, or maintained it involves intellectual discovery work from the human. We have had some "silver bullets" in the past, namely the third generation languages, the unified programming environments, and the time-sharing environments (Brooks, 1986) but Brooks did not see new bullets in the horizon. Brooks evaluated his claims nine years later (Brooks, 1995), and concluded that the main themes of his claims still hold true. Reuse is a way to improve efficiency and productivity but it is not an easy way. Further, there are no order-of-magnitude improvements waiting ahead. As a bright side of all of this, Brooks notes, is that we can finally concentrate on the essential problems and start making evolutionary improvements to our software processes, one step at a time.

Software engineering (SE) is concerned with practical application of scientific knowledge in the design and construction of computer programs and the associated documentation required to develop, operate, and maintain them (Boehm, 1976). Furthermore, SE must accommodate human, economic and programming concerns, and these concerns must be satisfied from both the product and process sides of the SE (Boehm, 1981, Pressman, 1992). Given this definition to software engineering we notice that it involves a number of different disciplines - computer science, management, psychology, design and economics, among other things (Freeman & Gaudel, 1991) - to solve problems in building software. In software engineering the concern is, on one hand, to build economically usable software systems, and, on the other, to manage the development process. (Boehm, 1981.)

Software includes all the artifacts that it takes to represent, i.e., document, it in machine or human readable form to the machines, workers, and stakeholders (Jacobson *et al.* 1999, Boehm, 1976). An *artifact* is a general term for any kind of information created, produced, changed, or used by workers in developing the system, e.g. UML diagrams and their associated text, user-interface sketches and prototypes, components, test plans, test procedures, and of course the code and programs (Jacobson *et al.* 1999). *Software development* is a disciplined way to produce systems, which deals with concepts, notations, processes, goals, and agents that are determined in a method used (see Koskinen, 2000).

As we stated above, software engineering has to consider the design of a software system. There is a need to model the design of the software system. According to Mathiassen *et al.* (1995) and Mathiassen and Stage (1992) there are different approaches that can be divided in two groups: specification based and exploratory based. Of the basic specification based approach, the waterfall model (Royce, 1970) is the prime example, whereas the exploratory approach is exemplified by the prototype approach (Gomaa, 1983). Third way to approach software development is a mixture of the two, which could be called evolutionary approach (see Sommerville, 1996). Examples of this third type include the spiral model (Boehm, 1986) and incremental models (see Graham, 1988, Gilb, 1988).

Initially the term life-cycle was used as a reference to the approaches mentioned above (see e.g., Davis *et al.* 1988, Acuña *et al.* 1999), but in the late 80's works by Osterweil (1987) and Humphrey (1989), among others, drew attention to the process aspect of software development. While the life cycle approach is concerned with the products that are produced during the development effort, the process view focuses on the production process. Although historically software development has been characterized as being product centered, researchers and developers have recently focused their efforts on the process dimension (Vasconcelos & Werner, 1998). A *process* is a written description of a course of action to be taken to perform a given task (Gibson, 1999).

Software development process (or just software process), is the set of activities needed to transform a user's requirements into a software system (Jacobson *et al.* 1999). The general intent of a software process is to coordinate individual activities so that they achieve a common goal (Cook & Wolf, 1998.) Software development process includes descriptions about how to produce software artifacts, and how to manage associated development and managerial activities (see Koskinen, 2000). Examples of software processes include design methods, change-control procedures, and testing strategies.

Software development most often takes place in projects. A software development project is an instance of the software development process (Jacobson *et al.* 1999), and the nature of software development makes it an "one-off" effort. Most of the work in software industry goes to research and construction of the first version of the software. The costs in software industry are mainly attributable to the development. Also, the maintenance can be considered to be an "one-off" effort because it involves same tasks as development and we do not make same kind of maintenance twice. The manufacturing itself is easy to do and can be done without any knowledge about software development.

Different approaches, or life cycles as they are called sometimes, tell us how software product evolves over the time. Software processes tell the order and the way different phases follow each other. However, life-cycles and processes give only a little help, if any, to software modelling. Neither do they tell what should be described and how these descriptions relate to each other. For these purposes we use software development methods.

The assumption in software engineering is that people use some sort of method to construct the software in the software development project (Yourdon, 1979). *Software development method* is a predefined and organized collection of techniques and a set of rules which state by whom, in what order, and in what way the techniques are used to achieve or maintain some objectives (Tolvanen, 1998). In many cases software development methods concern activities from the requirements analysis to testing. Most of the methods exclude, for example, maintenance, version and configuration control. One big part that Boehm (1976) stresses in SE is the management of the process. Management is not considered in most development methods, although this is changing nowadays.

We define a *technique* as a set of steps and rules which define how a representation of software is derived and handled using some conceptual structure and related notation (c.f. Tolvanen, 1998, Wieringa, 1998). Interpretation rules define the conceptual structure and describe, for example, meaning of the notation. Interconnection rules show the relations between different techniques. Techniques can be connected to each other in different ways. Heuristics defines steps and rules to produce different descriptions of the software in different phases of the development process. Some researchers consider, for example, data flow diagrams and class diagrams as techniques (Tolvanen, 1998).

Methods are explicit ways of structuring and rationalizing our thinking and action, involving both critical and creative thinking. In the field of information systems (IS) it is estimated that there are over 1000 methodologies (Jayaratna, 1994, Tolvanen, 1998, Koskinen, 2000). In the references mentioned these methods are described as information system development methods but most of them can also be considered as software development methods. Many of the recent methods consider themselves as software development methods. (see e.g., Jaaksi *et al.* 1999, Jacobson *et al.* 1999). Examples of structured methods are Modern Structured Analysis (Yourdon, 1989), and Jackson System Development (Jackson, 1983). Examples of Object-Oriented methods include OMT (Rumbaugh *et al.* 1991), OMT++ (Jaaksi *et al.* 1999), Catalysis (D'Souza and Wills, 1999), Unified Process (Jacobson *et al.* 1999) and Object-Oriented Software Engineering (Jacobson *et al.* 1992).

As stated above, SE seeks a way to improve the quality, pace, and utilization of resources in software development. Different approaches to software development, study of the software process, and all of the methods try to address some or all of these aspects. One specific way to increase productivity and quality in software development is to reuse existing software. In the field of software industry the idea was presented in 1968 when McIlroy (1976) argued that software could not be developed the same way it had been so far. He continued arguing that software development should be similar to computer construction where the computers are put together from components.

At present most of the reuse texts emphasize that reuse must be systematic, i.e. it must be planned, defined, and managed (Lim, 1998). The software reuse community has presented domain analysis as a way to foster reuse in software development. (Tracz, 1995.) Domain analysis takes place prior to actual development of software. A domain may be studied as a group of similar programs or as a business area (Wartik & Prieto-Díaz, 1992). With domain analysis, software developers try to identify general features in the domain in order to make them reusable later. During the development of a component the developers must also pay attention to documenting it. Documentation must be done in such a way that other developers in other projects can find and use the component.

In this study we concentrate on the reuse of the software components in software development. We examine the main limitations in current ways of doing component-based software development and we try to point out areas where improvements could be made. Also, we present some solutions to problems identified in creating and using components in software development. More specifically, this thesis shows a way to do domain analysis during software development and how resulting components should be documented in order to achieve a more efficient and more cost effective software development process without sacrificing product quality. At the end of the day, the user of the software could not care less about who created the individual solutions. His main concern is to find an end product combining these solutions to create something new and valuable to the user - faster, more efficiently and with more quality.

1.1.2 Research Questions

This study was conducted in PISKO project (Information Technology Research Institute/University of Jyväskylä, Finland). PISKO is funded by Technology Development Centre[1] and companies that participate in the project, namely Aplicom Oy, Honeywell Oy, TietoEnator Corp., Yomi Solution Ltd. (formerly Relatech Ltd.), and Republica Corp. All of these participants have made it possible to conduct this research.

The objective of the project was to introduce and develop component-based development approaches in the participating companies. The aim of the study is not to introduce a new component-based method, but rather to introduce techniques, which can be adopted by the companies. Furthermore, these techniques should not be biased towards any development or implementation technology. These limitations arise, on the one hand, because companies already have institutionalized some methods and many of their practices are tied up with these. On the other hand, companies use structured as well as object-oriented software development methods and languages.

The unifying idea behind the PISKO project could be phrased as follows: "How can software development be made more component-based and reuse-oriented?" From this general question we have drawn three more specific research questions, which we address in this study:

1. What are the specific limitations for reuse in current component-based software development methods?
2. How should reusable components be created and used in software development?
3. How should the components that are created be documented in order to make them reusable in other development projects?

The first question can be considered from two points of view. Firstly, we can examine current component-based development methods in order to find out what are the most visible problems in them. Secondly, we can study companies and their development methods in order to understand what is hindering the reuse of the components. By answering to these questions we can get more precise direction for our research efforts. We must remember the limitations to the solution that were set out at the beginning of this section when we search answer to the second question. This means that the solution must be tailored to the existing development method in a company. We look for some technique that enables a company to shift their thinking from one system to multiple systems and from the creation of a solution to one specific problem to a solution that is usable in other environments too. The third question is based on our belief that software is not only the code produced but also the documentation that supports the use of the program. An important area in the research of software engineering is the research of produced documentation. Also, we believe that creating components with proper documentation results in their successful and systematic reuse (see also Parnas, quoted in Brooks, 1995, p. 224).

[1] www.tekes.fi

1.1.3 Research Approach

Research method selection should be based on the research setting and the problem. What is found to be a relevant research problem by an individual researcher depends on his or her view of the world: what exists, i.e., ontology, and how the researcher believes that relevant information can be found, i.e., epistemology. If the researcher can be considered to be one factor in the formula of selecting a research approach then another factor is the discipline the researcher is in. Quite often a discipline has created its own understanding of what is perceived to be an appropriate approach towards research.

Information systems (IS) and software engineering (SE) fields are at a cross-section of number of disciplines. At the one end one can see for example cognitive and management sciences, which could be understood as "soft" or social sciences. At the other end there is computer science and mathematics, which could be considered as "hard" or natural sciences. Against this background it is no wonder that number of scholars suggest the use multi-methodological approach for studies in the IS and SE fields (cf. Nunamaker *et al.* 1991, Vidgen & Braa 1997, Mathiassen, 1998). As stated in the previous section, the aim of this study is to improve organization's software development practices to enable reuse. From a larger perspective this can be understood as an organizational change.

Mathiassen (1998) introduces research approach that he calls Reflective Systems Development (RSD). This approach intertwines both research and practice (Figure 1.1) and Mathiassen (1998, p. 81) describes it as follows:

> First, our understanding is based on interpretations of practice. Second, to support practice we simplify and generalize these interpretations and engage in design of normative propositions or artifacts, e.g. guidelines, standards, methods, techniques, and tools. Third, we change and improve practices through different forms of social and technical intervention.

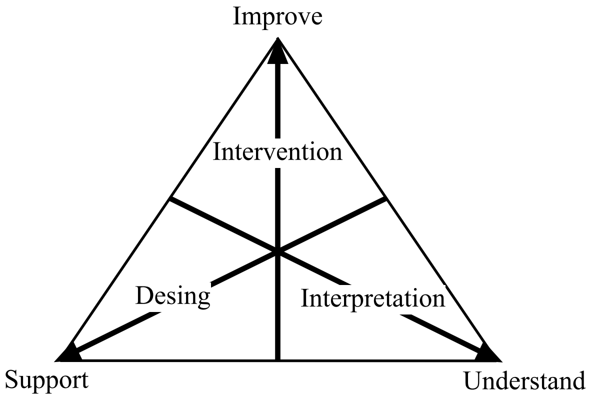


FIGURE 1.1 Research goals and activities involved in Reflective Systems Development. (Mathiassen, 1998, p. 82)

We follow this research approach in this study. According to it the first research activity is to understand current practice and interpret it. The aim of this phase is to find out what are the required enhancements in current practices. Here a researcher can use such methods as case studies, field studies, and sample surveys (c.f. Vidgen & Braa, 1997, Nunamaker *et al.* 1991). In this study we build our understanding and interpretation based on surveys to evaluate how current component-based development methods support reuse. Also, we do case studies in companies to understand their current software processes.

The second research activity is to create support for perceived problems or improvement areas. Aim of this phase is to create new methods or techniques that will improve current development process. This can also be considered as theory building. Research methods here are constructive and according to Nunamaker *et al.* (1991) its results are new ideas, concepts and conceptual frameworks, new methods, or models. We created a new technique to improve reuse in software development and sketched a model to document found components.

The third research activity, improvement, creates change in the organization and in its processes. In this phase we use the solutions that were created in the second phase and the objective is to prove their practicality. In this study we used the technique and model in an industrial setting in a project with TietoEnator Corporation[2]. TietoEnator is the largest Scandinavian software development organization with the staff of over 10 000 and annual net sales more than 1100 million euros.

1.1.4 Outline of Thesis

This thesis consists of an introductory chapter and six research papers, which follow the research approach. First, in the introductory chapter we introduce the research area and the research problem. The chapter gives an overview about the research area and clarifies some concepts and terms used. The focus of the chapter is in software engineering and reuse related areas. The chapter presents the results of the thesis. First there is a short summary of the papers and an outline about how they relate to research methodology. Also, the limitations of the study are discussed. The chapter closes with ideas for further study.

The rest of the thesis follows structurally the RSD approach. First we gain our understanding with three research papers. Here we refine the problems identified in connection with the more specific research questions and we improve our understanding of the problems. The first of the articles is "Evaluation of Component Based Software Development Methodologies". The article sheds light over the component-based approach to software development. We identify similarities between different methods to observe some underlying patterns. In addition, we identify some differences between methods to find out where consensus in the research field has not yet been achieved. The second article, "Use and Identification of Components in Component-based Software Develop-

[2] See www.tietoenator.com

ment Methods," takes a more close look into three component-based methods and studies their support of reuse. The third article, "A Modest but Practical Software Process Modeling Technique for Software Process Improvement", describes the process modeling approach we used in the companies we worked with. Here the aim is to introduce a lightweight and economic way to model software process so that its results can be used later when we introduce reuse-oriented software development process into the companies.

Next we design solutions for identified problems. Here we describe conceptual solutions to the problems that were identified while improving our understanding. We introduce a domain analysis method and a model to document software components. This is accomplished by two articles: "Using Hierarchy to Adapt Domain Analysis to Software Development" and "A Model to Document Software Components". The first article presents a domain analysis technique that does not only identify and define reusable components but also aids in the use of the components. Traditionally, domain analysis methods have focused solely on identification and definition of reusable components. The second article describes a model to document components. In this documentation frame three aspects of the reuse process are considered, namely the production, brokering and use of reusable components.

Finally, to complete our research cycle we present improvement which applies designed solutions. This is done with an article called "Adding Domain Analysis to Software Development Method" and it describes how we enhanced an existing software development method to include domain analysis. Also, we describe what kind of results we gained during this effort. We use the conceptual models that are presented in the papers "Using Hierarchy to Adapt Domain Analysis to Software Development" and "A Model to Document Software Components".

1.2 Software Reuse in Software Engineering

Reuse in software engineering was proposed at the same conference as the term software engineering was introduced (Krueger, 1992, see also Bauer, 1969 and McIlroy, 1976). So reuse and SE have co-existed for thirty years. But even before this conference Wilkes recognized the need to build a library of mathematical subroutines in 1949 to avoid having to rewrite them (Tracz, 1995). In 1968 McIlroy (McIlroy, 1976) introduced the idea of reusable components. McIlroy's components were comparable to standard, off-the-shelf components that are used in computer manufacturing. McIlroy's components would have been built by dedicated component factories. His components were source code components and were to be used in their original form in other programs. Unlike in other engineering disciplines, SE has only succeeded moderately, at the best, in reuse (Prieto-Díaz, 1994).

Software reuse is the systematic application of existing software artifacts during the process of building a new software system, or the physical incorporation of existing software artifacts in a new software system (Dusink and Katwijk, 1995). Here the term systematic means that the process of reuse is explicated and that the reusable elements are designed to be reused. Reasons for software reuse are the same as for SE. More programs should be created in shorter time, with smaller costs, with higher quality, and with fewer people. In the current economic situation the life cycle of a software product is shortening, i.e., new versions should be presented more often (cf. Bosch, 2000). This cannot be achieved unless great portions of the old version are reused in the next one.

1.2.1 Approaches to Software Reuse

Reuse in SE can be divided in composition and generation technologies (Biggerstaff & Richter, 1987). Composition techniques base reuse on the idea that components should be understood as atomic units and be reusable more or less unchanged. An external agent combines these components so that they fulfill any required functionality. Examples of this kind of components are design patterns (Gamma *et al.* 1995), code skeletons or frameworks (Fayad *et al.* 1999), software architectures (Tracz, 1995, Bosch, 2000) and of course binary components (e.g., Szyperski, 1997), among other things.

Generative technologies weave patterns or components into a program generator. These patterns are of two main types: patterns of code and patterns in transformation rules (Biggerstaff & Richter, 1987). Usually, in these techniques, software is defined in a higher level language and from this definition an application generator creates the required program. Patterns that are used to generate the software are modified to fulfill requirements. This way two generated patterns may vary greatly in the resulting program.

Frakes and Terry (1996) propose a faceted classification to identify different types of software reuse (Table 1.1). In the table each column specifies a facet.

In compositional reuse some portion of the products from the software development process are reused. This means that the project that creates the reusable part is not the same as the project which uses the component. Terms and definitions for a reusable part or component vary greatly (see e.g., Sametinger, 1997). It may be called an asset (Lim, 1998), artifact (Jacobson *et al.* 1999) or workproduct, among others. When using these terms the authors want to emphasize the fact that a component can be more than a code component. Jones (cited in Frakes & Terry, 1996, p. 417) identified ten different reusable aspects of software projects: architectures, source code, data, designs, documentation, estimates (templates), human interfaces, plans, requirements, and test cases. Krueger (1992) states that the types of reusable artifacts are not only fragments of code but can also be design structures, module-level implementation structures, specifications, documentation, transformations, and so on. Bosch (2000) defines a software component as a unit of composition with explicitly specified provided, required and configuration interfaces and quality attributes. Szyperski (1997) sees software components to be binary units of independent production, acquisition, and deployment that interact to form a functioning system.

TABLE 1.1 Types of Software Reuse. (Frakes & Terry, 1996, p. 418)

Development scope	Modification	Approach	Domain scope	Management	Reused entity
Internal (private)	White Box	Generative	Vertical	Systematic (planned)	Code
External (public)	Black Box (verbatim)	Compositional	Horizontal	Ad Hoc	Abstract level
	Adaptive (porting)	In-the-small			Instance level
		In-the-large			Customization reuse
		Indirect			Generic
		Direct			Source code
		Carried over			
		Leveraged			

The definitions vary greatly. Lim's assets include all kinds of products or byproducts of the software development process, including code, designs, test plans, and documentation, as well as knowledge and methodologies. Szyperski's software components are binary units of composition.

We define a (reusable) component as follows. A *reusable component* is a product of a software development process that can be used as part of a software product other than it was originally designed for. Additionally, a reusable component is documented in such a way that the documentation supports all reuse activities, namely component creation, brokering and consumption. In this thesis we quite often omit the suffix reusable when we talk about reusable components.

This definition means that a component is part of the end product of the software development process, for example, (part of) the defined analysis phase products. Also, we emphasize systematic approach by requiring that component's documentation must support reuse activities. Thirdly, we emphasize the point that reuse must take place in a different development project than where the component was created. We do not regard a component reusable unless these three aspects are satisfactorily taken into account. According to this definition, the software development process, development method, or project estimates are not reusable software components since they are not explicitly a part of the resulting end product.

1.2.2 Reuse Process

If an organization wants to facilitate reuse in its software development it should adapt reuse activities for its current software development process. Reuse-oriented software development creates software systems at least partly from existing assets in a systematic way. *Systematic software reuse* is the purposeful creation, management, support and reuse of reusable components (Jacobson *et al.* 1997). There exists some reuse process models (Jacobson *et al.* 1997, Lim, 1998, STARS, 1992, Karlsson, 1995) which address the features for reuse process. One of them is Lim's reuse model (this is presented more thoroughly in Chapter 3). In the Lim's model reuse-process is divided in four tasks: managing the reuse process, producing assets, brokering, and consuming assets. One problem with reuse methods and tools is that most of them describe only the use or creation of components, and further, reuse processes are not usually integrated to software development process (see Forsell *et al.* 2000, Rombach & Schäfer, 1994).

To give a basis for our discussion we present a model (Figure 1.2) where the creation, brokering, and consuming of the components are tied up with the software development process (see Lim, 1998, Griss *et al.* 1994). The model is coarse-grained and it only illuminates the most important aspects of the reuse-oriented software development, but for the needs of this study the model is sufficient. In the following sections we will cover each identified element of a reuse process more closely.

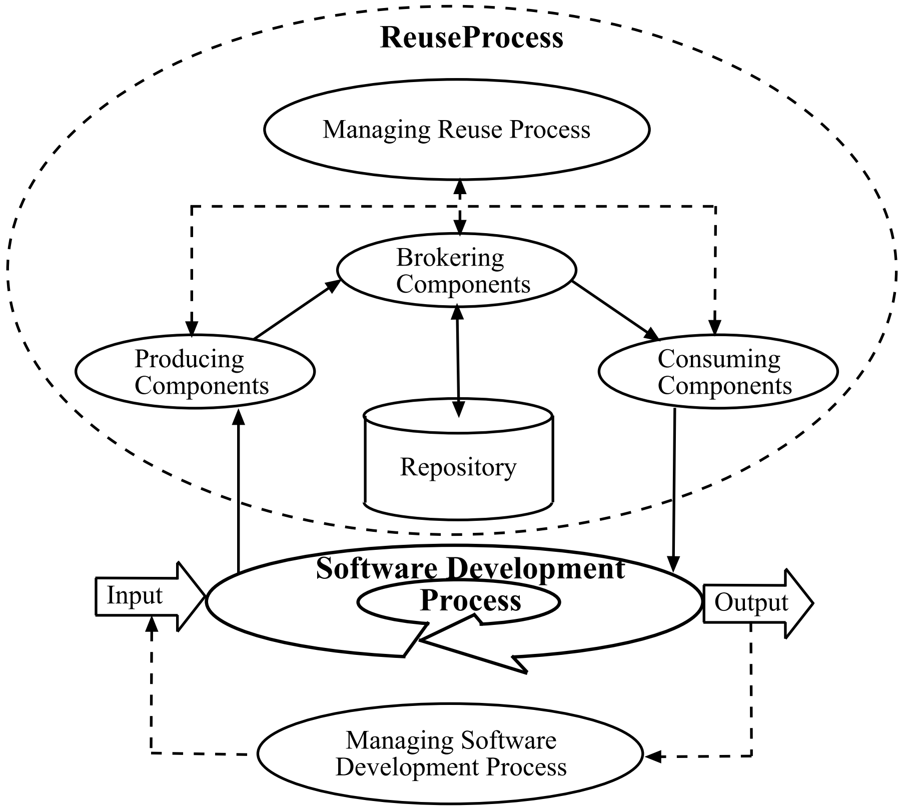


FIGURE 1.2 Model for reuse-oriented software development. In the model essential features of reuse processes are tied up to the software development process.

When a company wants to adapt reuse, it must plan how to integrate the reuse process into the software development process. One particular problem that must be solved is the dichotomy between the software development project which aims to produce one quality system and the reuse process which aims to support multiple development projects. This means that software reuse must be understood in much longer perspective than the development of a product. On the surface, the objectives of these two processes are quite different. But the bottom line is that both aim to produce quality software economically and efficiently.

Managing reuse-oriented software development

Management of reuse process is presented here first, because it should be considered first when starting a reuse program in organization. The foundation of systematic reuse lies in the planning of the whole reuse process and by defining what a company means by reuse and by a reusable component (see Griss *et al.* 1994). Before a reuse-oriented software development approach can be adapted, one has to know the current development process. In this section we look management from three points of view: management of the process, products and people.

One critical aspect in reuse-oriented software development is the management of the reuse process (see Lim, 1998, Griss *et al.* 1994, Jacobson *et al.* 1997). This means all the activities and procedures which are conducted in order to measure and control the reuse process. Different models and metrics (see Frakes & Terry, 1996 for more details) should be used in order to find out the effectiveness of the reuse process. By measuring the process we can, on one hand, reason what is good and works in the process, and on the other hand, we can find out flaws and points for improvement in our process. Also, measuring aids us in deciding whether to create certain components ourselves or acquire them from some other sources.

Managing the product means the management of components. With components, their quality plays a far more important role that their quantity. The reuser must trust these components as much as they would trust their own code (see Griss *et al.* 1994). The reuse level of the component, its usability, and the cost/benefit of using the component, among other things, must be measured.

Managing the people is our third viewpoint in reuse process management. Naturally we must know how people are engaging in reuse, that is how they use and produce components. To foster reuse, we must train people to different reuse techniques and at the same time we must create reward systems to make the practice of reuse more appealing. Most successful examples of reuse explicitly mention that developers were educated to use the reusable components (see e.g. Griss *et al.* 1994, Horowitz & Munson, 1984). With this kind of actions we can signal that the management is committed to reuse and that cultural change can start.

Producing components

Producing components could be called design for reuse or development for reuse (Griss *et al.* 1994). Component production involves such tasks as domain analysis and component creation.

Domain analysis is one of the key features in the road to use components (Prieto-Díaz, 1994). With domain analysis we can identify reusable components from a domain. There exists a number of methods or techniques to model and analyze domains. All of these try to achieve reusable abstractions from a group of software products or from a business domain.

Reusable components are created in a software development project. This means that components are a result of a software development effort. The same methods and techniques as in the production of conventional software products are used. Of course there is some additional criteria in the development of components and there may be a need to introduce new techniques, but, nevertheless, components are created in the same way as software systems.

New components come into existence also when old ones are enhanced or maintained (see Lim, 1998). Maintenance and enhancement effort can be divided in three categories, namely corrective, adaptive, and perfective (Lientz *et al.* 1978). Sometimes, no matter how extensive the tests and inspections made on the components, errors or flaws can be found in them. In these situations components must be fixed i.e. maintained. Maintenance is in these situations corrective. The old deficient component should in these situations always be replaced with a new, corrected component. Enhancement of the component can be in two forms: perfective or adaptive (see Schach, 1996). In perfective enhancement the component may get user enhancements, improved documentation or it can be made more efficient. In adaptive enhancement the component's ability to accommodate changes to data inputs and files or to hardware and software changes is improved. (cf. Lientz *et al.* 1978.)

Brokering components

Brokering components supports reuse across different projects. Brokering is one of the key elements when creating trust to existing components (see Tracz, 1995). This is why brokering is not only maintaining a component repository but it also includes tasks to validate and verify components that are put into a component repository. Lim (1998) identifies five distinct tasks to brokering components: assessing, procuring, certifying, adding, and deleting components.

Component assessment concerns the estimation of reusability for a component (Frakes & Terry, 1996). In assessing, all possible sources for component should be investigated and after potential components are found several factors should be explored: i.e. should the component be purchased or acquired, what is the quality of the documentation of the component, and how adaptable the component is. The cost/benefit analysis should be made (Lim, 1995) as well, of course.

Procuring components involves the rational decision about how components are obtained, i.e., is component bought or is it developed as an in-house project, among other possibilities.

Certifying assures that component meets the requirements, quality levels and includes necessary information (Lim, 1998). The components may come from various sources inside and outside the organization. Without certifying, the trust for components is hard to build.

Addition and deletion of a component to and from a repository must be taken into account. Adding components to a repository (or alike) involves cataloging, classifying, and describing it (Lim, 1998). Also the physical installation of the component into repository is done in this step. If a component is seen as irrelevant to an organization or it is determined that a component will no longer be valid, it should be deleted from the repository. To determine whether the component is useless or not must be done in accordance with the company's policies. When a deletion occurs, steps should be taken so that all the relevant parties are informed.

Consuming components

Much of the research in consuming components is done with the study of repositories. Most of these studies identify following steps for using components: find, understand, modify and integrate a component (see e.g. Taivalsaari, 1993, Prieto-Díaz, 1987). Further, another point that one must remember is that the documentation of a component is crucial in the consumption phase.

First, before we can start searching for the component we must identify the system we work with and the possible components that can be used within it. In this task the requirements for the system and thus for the component are identified. Two approaches are possible: reuse-enabled business approach and strategy-driven business approach. In reuse-enabled business, requirements may be altered according to the available components. In strategy-driven business the business-areas are partly chosen by available components, i.e. if we have readily available components that help to create certain kind of software only then we make decision to enter that particular market.

In the finding components phase, users of the components search repository if they expect to find required components there. The number of the relevant components that may be found is highly dependent on the capabilities of the repository and indexing schema used. Of course the number of components in the repository affects this, too.

Assessing components for the consumption phase requires the evaluation of the component found. Before we can evaluate a component and its suitability for the reuse we must understand it. After we have picked the components that can be used for a particular situation, only the best and most suitable one is selected. If components can not be found this creates an initial need for one and defined steps should be taken.

In adapting and modifying components two approaches are available: black-box reuse and white-box reuse. In black-box reuse components are used as they are. In white-box reuse components may be modified so that they fit more perfectly for use.

Integrating and/or incorporating components determines where and how components can be reused. This means that often components assume something about their environment and they cannot be used in incompatible environments. For example, if a code component is meant to be used with CORBA middleware, it is impossible to use it in DCOM environment without changes.

1.2.3 Domain Analysis for Software Reuse

One of the truisms in software reuse is that before you can reuse something you have to have something reusable (Tracz, 1995). It is not that simple to identify elements that will be reusable to developers. This has been painfully demonstrated by those who have attempted reuse approach in SE (Arango & Prieto-Díaz, 1991). Software reuse community has come up with one solution to address these difficulties. This is known as domain analysis. (Tracz, 1995.) Informally domain analysis (DA) could be defined as an analysis of some application area, leading toward some predefined goal (Arango & Prieto-Díaz, 1991). After DA is done in a certain domain we know what is available for reuse in that domain.

In the field of software engineering DA is seen as a prerequisite for successful reuse. DA developed from the work of Neighbors (1980). He was interested in automatically producing programs from existing components, and created an environment called Draco for this purpose (Neighbors 1980, 1989). Neighbors borrowed the idea of domain from automatic programming and from Balzer (1973) (see Neighbors 1980) and this resulted in the following definition to domain analysis: "A domain analysis is an attempt to identify the objects, operations and relationships between what domain experts perceive to be important about the domain." Here Neighbors quoted Balzer's (1973) definition as follows: "A model of the domain must be built and it must characterize the relevant relationships between entities in the problem domain and the actions in that domain." Neighbors argues that this is the definition of a problem domain. But if one reads more closely Balzer's (1973) article and carefully reads Neighbors definition, the definition itself does not define domain or problem domain, it defines what the model of the problem domain should include.

One difficulty in domain analysis is that there is no clear definition of the term domain. Wartik and Prieto-Díaz (1992) argue that domain can be understood from two points of view. First of all, a domain can be a group of similar applications or a program family (Parnas *et al.* 1989, Parnas, 1976). This point of view is related to application areas. The problems in a domain can be identified as based on the problems that applications solve. Second, a domain can be perceived as a business area. Here the domain is closely related with a business objective or business needs. Depending on which standpoint is taken, definition of a domain may become quite different.

We define domain here as based on Simos (1996, p. 423) who defines a domain to be:

> An abstraction that groups a set of software systems or some functional areas within systems according to a domain definition shared by a community of stakeholders. The domain can be considered to include not only the shared terminology and definitions, but the coherent body of knowledge about domain systems shared by that community.

And we add some more criteria from Arango and Prieto-Díaz (1991, p.13) that should be considered when selecting domains in SE:

1. deep or comprehensive relationships among the items of information are known or are suspected with respect to some class of problems,

2. there is a community that has a stake in solving the problems,

3. the community seeks software-intensive solutions to these problems, and

4. the community has access to knowledge that can be applied to solving the problems.

All DA approaches share the dichotomy between problem and solution (Wartik & Prieto-Díaz, 1992). When we have identified a meaningful domain we also have to define the solution space (Tracz, 1995). In software development the solution space is, by definition, software-intensive. This means that a solution is dependent on the software in one way or another.

The result of DA is a domain model. A domain model refers to those products that result from domain analysis. Again, the result depends on the notion of domain and on what were the goals for DA. The domain model can focus for example on repository, software specification, or process specification (Wartik & Prieto-Díaz, 1992). Naturally, results can also be used for variety of other objectives, e.g., gaining understanding and learning about the domain.

There are a number of domain analysis methods (see e.g. Arango & Prieto-Díaz 1991, Arango, 1994, Wartik & Prieto-Díaz, 1992). The ideal approach to DA should help software developers in every phase of the software development process and it should aid in selecting appropriate components for reuse (Wartik & Prieto-Díaz, 1992).

Arango (1994) evaluated DA methods that have a shared background, compositional software construction, and argued that they are equivalent. The evaluated methods use inductive generalization and classification as key steps in their processes, and they all can be mapped into The Common Process, as proposed by Arango (1994).

Unfortunately most of the DA methods do not consider themselves as integral part of software development but rather as a separate task. Exceptions to this are DA methods KAPTUR and Synthesis. (Arango, 1994.) The situation is not that much better if we look at the use of DA from the viewpoint of software development methods. McClure (1997) argues that domain analysis is missing from most of the currently widely used development methods, but it has been introduced in current CBD methods (Forsell *et al.* 2000). Shlaer and Mellor (1992) have presented a domain analysis method to be used in object-oriented software development. Their analysis method does not, however, contribute to reuse. In the current state-of-the-practice their domain modeling is comparable with analysis phase of OO methods. Recently developed component-based development methods (Jacobson *et al.* 1999, Jaaksi *et al.* 1999, D'Souza & Wills, 1999) do address domain analysis as a basis for successful systematic reuse. But in these methods DA is used more as a means to find components. Also, domain analysis is only briefly described and one gets the feeling that DA is not yet an integral part of these methods.

1.3 Summary of Papers

This thesis includes six research papers, which structurally follow the selected research approach. First we gain our understanding with three articles: "Evaluation of Component-Based Software Development Methodologies", "Use and Identification of Components in Component-Based Methods", and "A Modest but Practical Software Process Modeling Technique for Software Process Improvement". Although we use the word 'methodology' in the first title and 'method' in the second one, they are used here as synonyms. Here we aim at improving our understanding based on the interpretations of the practice. We want to understand the current state-of-the-art in the software development and this is done in the first two papers. Further, we also want to know how the organizations create their software, i.e., what is the state of the practice. A technique to gain this kind of knowledge is presented in the third paper. First three papers give us two points of view to our research area and we can interpret it more precisely. On the one hand, we know what the research in the area of component-based development suggests that we do. On the other hand, we gain knowledge about how a software development company creates its software.

Next we design support for the identified problems. This is done with two articles: "Using Hierarchies to Adapt Domain Analysis to Software Development" and "A Model for Documenting Reusable Software Components". The aim here is to support practice by creating solutions to the problems that are identified during the problem understanding. We concentrate on two specific problems, namely domain analysis and component documentation. Here we design a domain analysis model which can be integrated into an existing software development method. Further, we want to help in documenting components found so that they can be reused later on.

Finally we perform improvement. This is presented in the article "Adding Domain Analysis to Software Development Method". Here we intervene in an organization and try the created models in a software development project. Basically, we first integrate the hierarchical domain analysis method to an existing development method. Then we use this solution in practice and try to develop it further. We also use our component documentation model to the document components found.

In Figure 1.3 we show how these articles relate to the Reflective Systems Development approach (see section 1.1.3). In the following sections we describe each article in more detail.

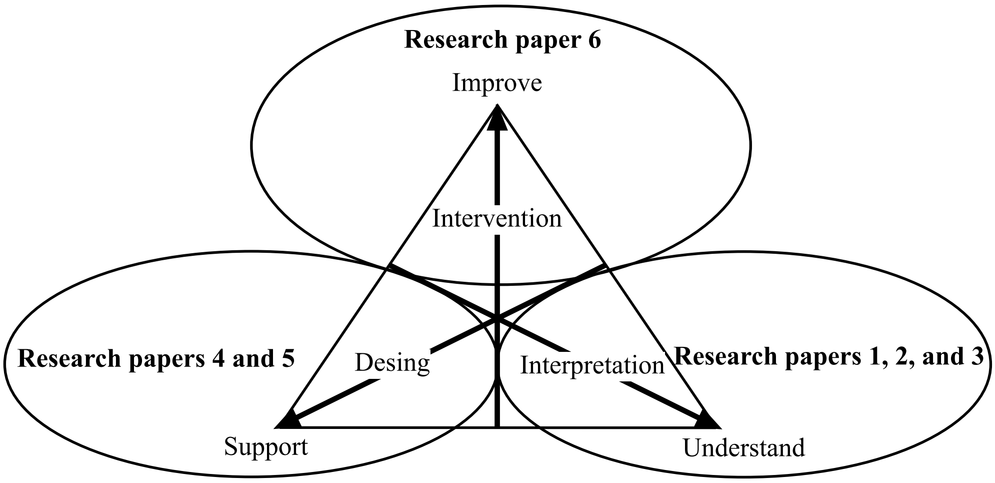


FIGURE 1.3 Research papers and their relation to the research approach.

1.3.1 Evaluation of Component-Based Software Development Methodologies

In an information society the users of different software are more aware of the opportunities of modern information technologies and require constantly new features for the software products. Software developers have to introduce new applications to the users, and maintain and create new features in the old software. As a result we need to find more efficient and effective ways to create software. Nevertheless, the software industry is bound to continue the current practices as long as the profits remain at the level they are today. Nowadays the pressure to produce more software is so high that there are not enough developers any more. One solution to these problems is to use component-based software development method.

Reuse does not just happen, it must be planned beforehand. For component-based development we need methods that support activities that are needed to create component-based software. In component-based software development methods we have to bring quality to component use and documentation. Needless to say that we have to pay attention to the context where components are used because without context analysis it is problematic to define what a component is and whether or not components can be successfully utilized. Our research objectives are (1) to find similarities in the methodologies, through which we attempt to expose the common features and aspects in component-based software development, and (2) to analyze differences between methodologies to point out areas where consensus has not been achieved.

We used survey and evaluation as research methods in this study. For evaluation we chose three known component-based development methods, namely Catalysis (D´Souza & Wills, 1999), OMT++ (Jaaksi *et al.* 1999), and Unified process (Jacobson *et al.* 1999). As an evaluation framework we used NIMSAD (Jayaratna, 1994) because it has a wide scope, it is not restricted to evaluation of any particular category of methodologies, it is practical, i.e., it has been used in several real-life cases, and it considers different use situations. Evaluation was done by carefully reading books that describe selected methods.

For the first research question the answer is that methods have many features that are similar. They are object-oriented and use Unified Modeling Language (UML) as the description language. This implies that UML is seen as an adequate means to model systems. Also, it seems evident that developer's experience is crucial in component use and definition.

For the second research question the answer is that differences between the methods lie in the development principles. Catalysis (D´Souza & Wills, 1999) emphasizes types and type models and stresses formalism. OMT++ (Jaaksi *et al.* 1999) seems a practical method, which provides a smooth transition from analysis to design. Unified Process (Jacobson *et al.* 1999) places particular stress on 'use cases', which guides the method users throughout the process. Another important aspect for Unified Process is the software architecture, which is created by the most influential use cases as early as possible.

The study contributes to the understanding of the current situation of component-based methods, and, in particular, it reveals a few areas where these and other methods could be improved. First of all, using components is left solely to the experienced developers. We suggest that information about components should be added in such a form that even less experienced developers could use the components. Secondly, evaluation of the development method use is neglected area in evaluated methods. Thirdly, problem formulation is done quite implicitly in these methods. Finally, implementation of the components is presented very briefly.

This article serves as the basis for our understanding of component-based development. We gain insights into component-based development and we have much firmer ground beneath us when we look for ways to improve the current state-of-the-practice. However, this study left open some bothersome questions about how components are explicitly identified and how they are used in the software development. The latter question is the topic of the second article.

1.3.2 Use and Identification of Components in Component-Based Software Development Methods

The component-based development tries to achieve reusability through the creation and use of components. In reuse the basic idea is to use a thing more than once. In this study we want, first of all, to find out how the component-based methods explicitly support the creation and use of components in the software development. Secondly, we want to expose the areas where these methods should be improved from the point of view of reuse.

As a research method we use survey and evaluation. This time we choose an 'ideal' reuse process as our evaluation framework. This we do because we believe that component-based development methods should support same activities that are seen necessary in the reuse process. Each method was evaluated in terms of its support for the activities and tasks of the 'ideal' reuse process model.

All evaluated methods place emphasis on producing components. Also, the coverage of the methods is quite similar according to the reuse model: they include domain modeling, production of components, and identification of a system. In practice, elements of component creation and use are intertwined, so it is noteworthy that the methods, actually, integrate the production of components into their use. The methods would be more usable if the production of components were distinctly separated from the use of components. Although components should be produced keeping the reuse aspect in mind, it is necessary to realize that the two tasks mentioned face different problems and require different solutions and need to be managed as separate processes. Because the tasks of producing and using components are intertwined, it makes the methods more complicated and decreases their usability from the reuse point of view. As a result, no or little information on the components is saved to assist in the further use of those components. It is obvious that in many cases the software people do not even know what is the necessary information that helps in finding a suitable component.

Are the evaluated methods component-based? They are, in the sense that they aim at well-structured architectures, where the purpose is to ease the manageability of the development process and the software. However, they have several weak areas that need to be improved if it is desired to gain the full benefit from the use of components. First, the methods should put more emphasis on the sub-processes of the component use. Secondly, methods should support the use of components from a multi-purpose perspective rather that a single application perspective. Thirdly, the methods should include tools to collect relevant information on the components to be saved into a repository that could then be effectively used when searching for suitable components.

To guide further research concerning the component-based methods we see the following research questions as relevant:

1. How to document components of different levels so that people who are not domain experts could use them?

2. What kind of a repository would be the most valuable in supporting the reuse process?

3. How the different reuse-oriented activities (e.g. managing the reuse infrastructure, producing, brokering, and consuming reusable assets) can be adapted to a software development process?

This paper together with the first one gives us good understanding of the current component-based methods, and together they help us to interpret the current situation. The first paper focuses on the methods at a reasonably general level while the second one focuses on the creation and use of the components in software development. This paper also gives us hints about what areas are neglected in current CBDs. One area that needs support is the need of expertise in reuse. This need should be decreased so that people without extensive knowledge about application area and readily available components could practice reuse. Secondly, domain analysis is seen as important and methods do cover that but not sufficiently. Domain analysis has little exposure and it is not presented in the central role that we expect it to be. Thirdly, documentation of the components is in its infancy and there is not much support for this area. In later papers where we present the domain analysis method proposed as a result of this research and a model for documenting components we attack these problems directly. But before we can propose any solution for a company, we have to understand its current software development process. This is done in the next paper.

1.3.3 A Modest but Practical Software Process Modeling Technique for Software Process Improvement

In order to improve the current software development process, one must know what the process is like. Processes can be assessed with various models, e.g., CMM (Paulk *et al.* 1993, Paulk *et al.* 1995), SPICE SPICE, 2001, and ISO9001 (Kehoe & Jarvis, 1996). The problems with these models are that they require extensive knowledge about the models and lots of resources from the target organization. Furthermore, processes are evaluated according to an ideal process, which may not be suitable for a particular organization. In our particular case the ideal process should include reuse processes, too. In addition, there are very few techniques that include a process description for assessment, a modeling guide, and a documentation guide in a single package. In this paper we present such a technique.

As a research method we used a questionnaire to determine the suitability of the presented technique within industrial software organizations. We conducted twelve case studies in four organizations. We used the presented technique on three distinct occasions. First, we modeled the overall software development process in those organizations. Second, we modeled the reuse of the components in that process. Third, we produced a process model about the testing in that process. The questionnaire was sent to the experts in organizations who attended the modeling sessions. Based on this questionnaire we evaluate the presented technique and compare results with experience published about CMM-based appraisals.

According to the answers, the technique is suitable for modeling the software process, and it identifies the points of improvement and problems with current process. It does not, however, give any order in which those improvements should take place. The target organizations have implemented some of the suggested improvements but not in the scale expected. When comparing the results from our technique with CMM-based appraisals it is interesting to note that the results are roughly comparable. This technique, however, requires less effort from target organizations.

The technique presented is usable in modeling software processes and it gives information about the points of improvement. Organizations that want to gain deeper understanding about their current process can use the technique. Also, the technique is usable by consultants who have understanding about software development. In the future, however, the technique should be improved so that it would enable enumeration of problems and points of improvement and give more direct advice about how to initiate improvements.

This paper presents a way of modeling and reasoning about an organization's current software process. The results that we gained during the process modeling were used as the basis for the improvements that are presented in the following papers. From the point of view of our research framework this paper presents a method that can be used to interpret current situation from the point of view of the organization.

1.3.4 Using Hierarchies to Adapt Domain Analysis to Software Development

Domain analysis (DA) is a prerequisite for successful component reuse and it aims to identify possible components from the application domain, among other things. DA is a process through which information used in software development is identified, captured, and organized with the purpose of making it reusable when creating new systems. Traditional development methods focus on one application but domain analysis focuses on classes of applications.

Current component-based development methods (Jaaksi *et al.* 1999; Jacobson *et al.* 1999; D´Souza & Wills, 1999) use DA as one part of the development work (Forsell *et al.* 2000). Unfortunately, DA is only briefly introduced and it is used to identify possible components inside one application. The components in these methods provide a means to manage development work. Also, the components are seen as a way to help maintenance later on, so that changes will not propagate all over the software.

Our problem here is twofold. On one hand, a development method needs to use components as available resources, not only as a means to control the development or maintenance work. On the other hand, DA needs to be part of all of the company's software development projects, and not residing only inside narrow, highly specified, domains. Based on these observations we specify our research question as follows: "How should the domain(s) be analyzed so that all solutions readily available could be part of the resulting software solution, and that results could be produced and used in everyday software development?"

We assume in this paper that domain analysis can be used to answer to this research question. The research method here is constructive and it aims at theory building in a form of a technique to perform domain analysis.

In the paper we present a technique called hierarchical domain analysis (HDA). In HDA all the software a company produces is seen as a domain, and by dividing this 'super' domain into sub-domains, we can use results from the domain analysis in all the software development projects the company performs. In the paper we show the process of how to perform HDA.

This paper solves some of the problems that we identified in the previous papers and aims to create support for the software development process. But it is not enough that we present a technique to find and use components. There must also exist guidelines to document found components so that they can be found in subsequent projects, i.e., we must also support the brokering of the components. This is the topic of the next paper of this thesis.

1.3.5 A Model for Documenting Reusable Software Components

The effective reuse of the software components needs an effective means for documenting and communicating many kinds of related information among several stakeholders in the reuse process. The existing models for component documentation (e.g., Karlsson, 1995, NATO, 1993, Sametinger, 1997) plainly highlight the information content required, without much attention, let alone a theoretical basis, for the processing and communication of documents by and among the stakeholders. Our research problem here is that we want to find a theoretically based model for component documentation.

The research method we use in this paper is theory building. Our aim is to present a model for component documentation. We take Lim's reuse model (Lim, 1998) as the starting point and we look at it through the lens of genre theory (Yates & Orlikowski, 1992, Bazerman, 1994, Orlikowski & Yates, 1998) .

Our model for component documentation supports the reuse process. The documentation model has two major parts: a reusable part and a part that enables the reuse. The reusable part is the component itself and it becomes a part of the final software product. The second part supports the reuse activities that are going on when reusing the components, namely the brokering, and consuming of the components, and the management of the reuse process.

The documentation model presented stays at an abstract level and it should be refined at least in two ways. First of all, it should be refined to address the specific questions that we come across when we are documenting components in different abstraction levels, i.e., what should be addressed in the design level differently from that in the code level, for example. Secondly, it should address the differences in development methods.

This article addresses the questions that we presented in the second article. Within the thesis as a whole, this paper supports the documentation of the found components. Further, it addresses some basic questions in the rather neglected area of component documentation. In the last two papers we have suggested some solutions to the problem of supporting component reuse in software development. In the final paper we use these results in a software development project in an industrial setting.

1.3.6 Adding Domain Analysis to Software Development Method

When a company wants to use and create components in software development, one critical success factor is the use of domain analysis (DA). We have presented a domain analysis method called Hierarchical Domain Analysis (HDA) and a documentation model for reusable components in papers four and five, respectively. The objective of this paper is to try HDA in practice and to find out what parts of it should be enhanced. Also, we try to use the presented documentation model. The research objective implies that first we integrate HDA into an existing development method and then find out its practicality in a software development project.

We use the action case research method (Vidgen & Braa, 1997) in this paper. We first integrate HDA (Forsell, 2001) into TietoEnator's in-house development method Tieto Object. After this we use the HDA technique in a pilot project. We evaluate our experience using Leavitt's diamond model (Leavitt, 1965).

The integration of HDA, in our opinion, was successful in three major ways. First of all it produced reusable business components. Secondly, it showed that HDA is an approach that is useful. Thirdly, we gained understanding of the refinement of HDA and we now have experience and material to educate developers within TietoEnator Corporation.

We want to point out four other contributions this study makes. First, here we have shown how we integrated HDA into Tieto Object. We believe that our comments about the integration should be taken into account whenever practitioners intend to integrate DA into a development process. Developers must first identify any need for reuse and objectives for it. Then one must determine the kinds of components to be used. Only after this, one can select a domain analysis method. We also listed the steps we took during the local method development. Second, we showed that with HDA one can find reusable business components and get an idea about where in the domain they can be reused. Third, we presented how to document and model the results. Fourth, we found out what kinds of roles are needed to take advantage of the results.

The main limitations in this study are that it concentrates on one company and on one large project. Also, we suspect that modeling an organization's hierarchy might not be the best way to model hierarchical domain. The modeling of the hierarchy might be better when reflecting the mission of an organization rather than the existing chain of command.

This research paper completes our research cycle. Problems identified and solutions sketched are now tried out in practice successfully.

1.3.7 About the Joint Articles

The first two articles “Evaluation of Component-Based Software Development Methodologies” and “Use and Identification of Components in Component-Based Software Development Methods” were co-authored by Veikko Halttunen and Jarmo Ahonen. In both of these articles the research was designed with J. Ahonen and V. Halttunen. I conducted the research and wrote the first drafts of the articles. The consequent versions of the articles were written jointly with V. Halttunen and J. Ahonen.

The third joint article “A Modest but Practical Software Process Modeling Technique for Software Process Improvement” was written with J. Ahonen and Sanna-Kaisa Taskinen. The research design was done together with J. Ahonen. S-K. Taskinen did the empirical part of the work. I, Ari Häkkinen and Esko Hakulinen, designed the first version of the process modeling technique. Later on, J. Ahonen, E. Hakulinen, V. Halttunen, and I further refined the technique. The first draft of the article was written by me. J. Ahonen and I wrote the consequent versions of the article.

The fourth joint article “A Model for Documenting Reusable Software Components” was written with Tero Päivärinta. I was responsible for designing the research. T. Päivärinta wrote the background about the genre-theory while I was responsible for the reuse perspective. T. Päivärinta and I conducted the research equally.

1.4 Limitations of This Study

The limitations of this study are grouped in two categories: premises and limitations in the research. The premises that this study relies on are that:

1. reuse must be systematic,
2. systematic reuse can happen with components,
3. domain analysis is crucial for successful component reuse, and
4. only by documenting the components can we assure successful reuse.

The first premise requires that we must focus on reuse process and that we must define and plan reuse beforehand. Our systematic approach requires that we have to know what kind of reuse we are planning to do. The second premise limits our view to the reuse of the components. We acknowledge that there are other types of reuse that are successful and can be used to achieve same sort of goals as in this study. The third premise requires that reuse process and the successful reuse of components must rely on domain analysis. As stated above, the domain analysis is the only technique that software reuse community have presented to foster reuse in software development. We take this as granted. The fourth premise requires us to focus on the documentation in the software reuse. When components are identified and created, we do not consider them reusable unless they are properly documented for the possible users.

We focused only on domain analysis and component documentation as ways to improve current way of developing software. There are other ways as well. For example, there are generative reuse approaches which were wholly excluded in this study. Nevertheless, we believe that if a company wants to create components and use these components later on, domain analysis is one of the first steps that it should take, and if a company creates components they should be documented somehow.

There are number of limitations in the conduct of the research, too. The limitations are:

1. the main results rely on one action case study,
2. the focus of the reuse is from the developer's point of view,
3. the research cycle was conducted only once, and
4. there is no way to model the way business aspect services are dependent on specific software aspect solutions.

The major limitation in the conduct of the research is that the final paper, where the solutions are tried out in practice, concerns one company only. Although we argue that integration of any technique is situation bound and depends on the environment, one can always claim that in this study we have tested our solutions in one company only. Based on this single case study we can not determine if HDA is suitable, for example, in modelling embedded software.

The second limitation is that we have focused only on the reuse in software development and our focus is limited to the software developer's point of view. The management of the reuse process and the software development process is not considered. This does not mean that we do not see these issues as important. We are first to admit that the road towards successful reuse starts with the support of the upper management and reuse should be supported throughout the organization.

The third limitation points out, that RSD can be viewed as a continuous improvement cycle. Here we have completed the research cycle once and now we would have good basis to do it again.

The fourth limitation focuses to the limitation of the hierarchical domain analysis. Currently there is no way to model how certain business aspect's service is dependent on particular software aspect's solutions. This is not important when we model design pattern type components but if we model code components then there should be a way to see instantly what kind of system and middleware software must exist. In its current form HDA requires component user to read the documentation of a component to determine what kind of software any particular component requires.

1.5 Conclusions

This study is about how to improve component reuse in software development. More specifically, this study explores how to support reuse-oriented software development. Our research questions have been:

1. What are the specific limitations for reuse in the current component-based software development methods?
2. How should reusable components be created and used in software development?
3. How the components that are created should be documented in order to make them reusable in other development projects?

The answer to the first question is that current component-based development methods aid more in the creation of the components, and not in their use. Also, the components are seen as a means to manage software development process, because with the components the different parts of the software can be created and maintained independently, and maintenance and enhancements do not propagate to other parts of the software.

Our second question is derived from the problems we encountered with while answering to the first research question. We have proposed that domain analysis could also help in using components, not only in creating them. With the help of the hierarchical approach to domain analysis, software developers can find components in a variety of domains and use results from one domain across different domains.

The third research question can also be derived from the answer to the first research question. One limitation that all the evaluated methods have is that they do not give any specific support to documentation of the components. We proposed a model that would help a company to create documentation for their reusable components.

We can see that this study makes four major contributions. First, it evaluates three known component-based software development methods. Any company that wish to start using component-based development method can use the results of the evaluations. Results from these evaluations should also help the research community in method development in weak areas identified.

The second contribution is a technique to model companies' software processes practically and economically. The technique can be used as a first step when a company wants to initiate process improvement.

The third contribution is the hierarchical domain analysis that gives also guidance in the use of components while current domain analysis methods focus only on finding and defining components. Hierarchical domain analysis can be used by companies who want to make their current way of creating software more reuse-oriented.

The fourth contribution is the documentation model for reusable components. When companies want to take a systematic approach to the software reuse and when these companies are designing more specific component documentation or a repository description, they can use this documentation framework. Also, the documentation model should be interesting for further study by researchers, because this area needs further improvement.

The results that are presented in this thesis are not biased towards any specific software development method or any implementation technique (i.e. procedural or object-oriented). The results should be usable by companies that wish to start using a reuse-oriented approach to their software development. Nevertheless, one must bear in mind, that the case study presents results from a single company, which creates tailored and packaged software for its customers. The software thus created is business-oriented.

In the future, more emphasis should be placed on the empirical testing of the results. This means that the results, especially HDA and the component documentation model, should be used in a variety of companies and real-life software development projects. In this way, both the HDA and the documentation model can be refined further.

References

Acuña, A., Lopez, M., Juristo, N., Moreno, A., A Process Model Applicable to Software Engineering and Knowledge Engineering. International Journal of Software Engineering and Knowledge Engineering, Vol. 9, No. 5, 1999, pp. 663-687.

Arango, G., Domain Analysis Methods. In Schäfer, W., Prieto-Díaz, R., Matsumoto, M. (eds.), Software Reusability. Ellis Horwood, 1994, pp. 17-49.

Arango, G., Prieto-Díaz, R., Introduction and Overview: Domain Analysis Concepts and Research Directions. In Prieto-Díaz, R., Arango, G. (eds.), Domain Analysis and Software Systems Modeling, IEEE Computer Society Press Tutorial, Los Alamitos, CA, 1991, pp. 9-32.

Balzer, R., A Global View of Automatic Programming. Proceedings of the Third Joint Conference on Artificial Intelligence, SRI International, August 1973, pp. 494-499.

Bauer, F., In Naur, P., Randell, B.(eds.), Software Engineering: A Report on a Conference Sponsored by the NATO Science Committee, NATO, 1969, Cited in (Pressman, 2000). NATO Conference held in Garmish, Germany, 1968.

Bazerman, C., Systems of Genres and the Enactment of Social Intentions, in Freedman A., Medway, P. (Eds.), Genre and the New Rhetoric ,Taylor & Francis, London, 1994, pp. 79-101.

Biggerstaff, T., Richter, C., Reusability Framework, Assessment, and Directions. IEEE Software, Vol. 4, No. 2, March 1987, pp. 41-49.

Boehm, B., Software Engineering. IEEE Transactions on Computers, Vol. C-25, No. 12, December 1976, pp. 1226-1241.

Boehm, B, Software Engineering Economics. Prentice-Hall Inc., Upper Saddle River, NJ, 1981.

Boehm, B., A Spiral Model of Software Development and Enhancement. ACM SIGSOFT Software Engineering Notes, Vol. 11, No. 4, August 1986, pp. 14-24.

Bosch, J., Design & Use of Software Architectures. Addison-Wesley, 2000.

Brooks, F., No Silver Bullet - Essence and Accidents of Software Engineering. In Kugler, H.-J. (ed.), Information Processing '86, Elsevier North-Holland, 1986, pp. 1069-1976.

Brooks, F., The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition. Addison-Wesley Longman, Inc., 1995.

Cook, J., Wolf, A., Discovering Models of Software Processes from Event-Based Data. ACM Transactions on Software Engineering and Methodology, Vol. 7, No. 3, July 1998, pp. 215-249.

Davis, A., Bersoff, E., Comer, E., A Strategy for Comparing Alternative Software Development Life Cycle Models. IEEE Transactions on Software Engineering, Vol. 14, No. 10, October 1988, pp. 1453-1461.

D'Souza, D., Wills, A., Objects, Components, and Frameworks with UML: The Catalysis Approach, Addison-Wesley, 1999.

Dusink, L., van Katwijk, J., Reuse Dimensions. Proceedings of the the 17th international conference on software engineering on Symposium on software reusability , Seattle, Washington, April 28-30th 1995, Software Engineering Notes, Volume 20, No. SI, August 1995, pp. 137-149.

Fayad, M., Schmidt, D., Johnson, R., Building Application Frameworks: Object-Oriented Foundations of Framework Design. John Wiley & Sons, Inc., 1999.

Frakes, W., Terry, C., Software Reuse: Metrics and Models, ACM Computing Surveys, Vol. 28, No. 2, June 1996, pp. 415-435.

Freeman, P., Gaudel, M.-C., Building a Foundation for the Future of Software Engineering. Communications of ACM, Vol. 34, No. 5, May 1991, pp. 31-33.

Forsell, M., Using Hierarchies to Adapt Domain Analysis to Software Development. in Sein, M., Munkvold, B., Ørvik, T., Wojtkowski, W., Wojtkowski, W. G., Zupančič, J. (eds.), Contemporary Trends in Systems Development. Papers presented at ISD2000, the Ninth International Conference on Information Systems Development: Methods and Tools, Theory and Practice, August 14-16, 2000, Kristiansand, Norway. Kluwer Academic/Plenum Publishers, New York, NY, 2001, pp. 105-118.

Forsell, M., Halttunen, V., Ahonen, J., Use and Identification of Components in Component-Based Software Development Methods. Frakes, W. (ed.), Software Reuse: Advances in Software Reusability. Proceedings of the 6th International Conference, ICSR-6, Vienna, Austria, June 2000. Lecture Notes in Computer Science 1844. Springer-Verlag, 2000.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., Design Patterns: Elements of Reusable Software. Addison-Wesley, 1995.

Gibson, R., Software Process Modeling. In McGuire, E. (ed.), Software Process Improvement: Concepts and Practices, Idea Group Publishing, Hershey, PA, 1999, pp. 1-16.

Gilb, T., Principles of Software Engineering Management. Addison-Wesley Publishing Company, 1988.

Gomaa, H., The Impact of Rapid Prototyping on Specifying User Requirements. ACM SIGSOFT Software Engineering Notes, Vol. 8, No. 2, April 1983, pp. 17-28.

Graham, D., Incremental Development: Review of Non-monolithic Life-cycle Development Models. Information and Software Technology, Vol. 31, No. 1, January/February 1988, pp. 7-20.

Griss, M., Favaro, J., Walton, P., Managerial and Organizational Issues - Starting and Running a Software Reuse Program. In Schäfer, W., Prieto-Díaz, R., Matsumoto, M. (eds.), Software Reusability. Ellis Horwood, 1994, pp. 51-78.

Horowitz, E., Munson, J., An Expansive View of Reusable Software. IEEE Transactions on Software Engineering, Vol. 10, No. 5, September 1984, pp. 477-487.

Humphrey, W., Managing the Software Process. Addison-Wesley Publishing Company, Inc. 1989.

Jaaksi, A., Aalto, J.-M., Aalto, A., Vättö, K., Tried & True Object Development: Industry-Proven Approaches with UML. Cambridge University Press, 1999.

Jackson, M., System Development. Prentice Hall, 1983.

Jacobson, I., Booch, G., Rumbaugh, J., The Unified Software Development Process. Addison-Wesley, 1999.

Jacobson, I., Christerson, M., Jonsson, P., Overgaard, G., Object-Oriented Software Engineering: A Use Case Driven Approach. ACM Press, New York, NY, 1992.

Jacobson, I., Griss, M., Jonsson, P., Software Reuse: Architecture, Process and Organization for Business Success. Addison-Wesley, 1997.

Jayaratna, N., Understanding and Evaluating Methodologies. McGraw-Hill Book Company, Maidenhead, England, 1994.

Karlsson, E., Software Reuse: a Holistic Approach. John Wiley & Sons Ltd., Chichester, England, 1995.

Kehoe, R., Jarvis, A., ISO 9000-3: A Tool for Software Product and Process Improvement, Springer-Verlag, New York, 1996

Koskinen, M., Process Metamodelling: Conceptual Foundations and Application. Ph.D. Thesis, Jyväskylä Studies in Computing, No. 7, University of Jyväskylä, 2000.

Krueger, C., Software Reuse. ACM Computer Surveys, Vol. 24, No. 2, June 1992, pp. 131-183.

Leavitt, H., Applied Organizational change in industry: structural, technological and humanistic approaches. In March, J. (ed.), Handbook of Organizations, Rand McNally & Company, 1965, 3rd printing 1970.

Lim, W., Effects of Reuse on Quality, Productivity, and Economics. IEEE Software, Vol. 12, No. 5, September 1995, pp. 23-30.

Lim, W., Managing Software Reuse. Prentice Hall Inc. Upper Saddle River, NJ, 1998.

Lientz, B., Swanson, E., Tompkins, G., Characteristics of Application Software Maintenance. Communications of the ACM, Vol. 21, No. 6, June 1978, pp. 466-471.

Mathiassen, L., Reflective Systems Development. Scandinavian Journal of Information Systems, Vol. 10, No. 1&2, 1998, 67-117.

Mathiassen, L., Seewaldt, T., Stage, J., Prototyping and Specifying: Principles and Practices of a Mixed Approach. Scandinavian Journal of Information Systems, Vol. 7, No. 1, 1995, pp. 55-72.

Mathiassen, L., Stage, J., The Principle of Limited Reduction in Software Design. Information, Technology an People, Vol. 6, No. 2-3, 1992, pp. 171-185.

McClure, C., Software Reuse Techniques: Adding Reuse to the Systems Development Process. Prentice Hall PTR, New Jersey, 1997.

McIlroy, M., Mass-produced Software Components. In Naur, P., Randell, B., Buxton, J. (eds.), Software Engineering Concepts and Techniques, Proceedings of the NATO Conferences, Petrocelli/Charter, 1976, p. 88-89. NATO Conference held in Garmish, Germany, 1968.

NATO, NATO Standard for the Development of Reusable Software Components, Volume 1 (of 3 Documents), (http://www.asset.com/WSRD/abstracts/archived/ABSTRACT_528.html), (accessed 1 June 2000), 1993.

Neighbors, J., Software Construction Using Components. Ph.D. thesis, TR-160, University of California, Irvine, ICS Department, 1980.

Neighbors, J., Draco: A Method for Engineering Reusable Software Systems. In Biggerstaff, T., Perlis, A. (eds.), Software Reusability Volume I: Concepts and Models, ACM Press, New York, NY, 1989, pp. 295-319.

Nunamaker, J., Chen, M., Purdin, T., Systems Development in Information Systems Research. Journal of Management Information Systems, Vol. 7, No. 3, 1991, 89-106.

Orlikowski, W.J., Yates, J., Genre Systems: Structuring Interaction through Communicative Norms, Sloan School of Management Working Paper #4030, MIT, http://ccs.mit.edu/papers/CCSWP205), (accessed 12 July 1999), 1998.

Osterweil, L., Software Processes Are Software Too. Proceedings of the 9th International Conference on Software Engineering, Monterey, California, USA, May 1987, IEEE Computer Society Press, USA, 1987, pp. 2-13.

Parnas, D., On the Design and Development of Program Families. IEEE Transactions on Software Engineering, Vol. 5, No. 2, March 1976, pp. 1-9.

Parnas, D., Clemens, P., Weiss, D., Enhancing Reusability with Information Hiding. In Biggerstaff, T., Perlis, A. (eds.), Software Reusability, Volume1: Concepts and Models. ACM Press, New York, NY, 1989, pp. 141-158.

Paulk, M., Curtis, W., Chrissis, M., Weber, C., Capability Maturity Model for Software, Version 1.1. Technical Report, CMU/SEI-93-TR-24, DTIC ADA263404, 1993.

Paulk, M., Weber, C., Curtis, B., Chrissis, M. 1995. The Capability Maturity Model: Guidelines for Improving the Software Process, Addison-Wesley, Reading Massachusetts, 1995.

Pressman, R., Software Engineering, Fourth Edition. Addison-Wesley, 1992.

Pressman, R. (Adapted by Ince, D.), Software Engineering: A Practitioner's Approach, European Adaptation, Fifth Edition. Addison-Wesley, Maidenhead Berkshire, 2000.

Prieto-Díaz, R., Historical Overview. In Schäfer, W., Prieto-Díaz, R., Matsumoto, M. (eds.), Software Reusability. Ellis Horwood, 1994, pp. 1-16.

Prieto-Díaz, R., Freeman, P., Classifying Software for Reusability. IEEE Software, Vol. 4, No. 1, January 1987, pp. 6-16.

Rombach, H., Schäfer, W. Tools and environments. In (Schäfer, W., Prieto-Díaz, R., Matsumoto, M. (eds.), Software Reusability. Ellis Horwood, 1994, pp. 113-116.

Royce, W., Managing the Development of Large Software Systems: Concepts and Techniques, 1970 WESCON Technical Papers, Western Electric Show and Convention, Los Angeles, August 1970, pp. A/1-1 - A/1-9.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W., Object-Oriented Modeling and Design. Prentice Hall, Englewood Cliffs, NJ, 1991.

Sametinger, J., Software Engineering with Reusable Components, Springer, 1997.

Schach, S., Classical and Object-Oriented Software Engineering, Third Edition. McGraw-Hill, 1996.

Shlaer, S., Mellor, S., Object Lifecycles: Modeling the World in States, Prentice-Hall, Yourdon Press Computing Series, 1992.

Simos, M., Organization Domain Modeling (ODM) Guidebook Version 2.0, Informal Technical Report for Software Technology for Adaptable, Reliable Systems (STARS), STARS-VC-A025/001/00, 14 June 1996.

Sommerville, I., Software Process Models. ACM Computing Surveys, Vol. 28, No. 1, March 1996, pp. 269-271.

SPICE, http://www.sqi.gu.edu.au/spice/, visited 31.8.2001.

STARS, STARS Conceptual Framework for Reuse Processes (CFRP), Volume I: Definition, Version 3.0, STARS-VC-A018/001/00, Informal Technical Report, 1992. (http://www.asset.com/WSRD/ASSET/A/495//ASSET_A_495.tar.gz), (accessed 1 June 2000).

Szyperski, C., Component Software. Addison-Wesley, 1997.

Taivalsaari, A., A Critical View of Inheritance and Reusability in Object-oriented Programming. Ph.D. Thesis, Jyväskylä Studies in Computer Science, Economics and Statistics, No. 23, University of Jyväskylä, 1993.

Tracz, W., Confessions of a Used Program Salesman: Institutionalizing Software Reuse. Addison-Wesley Publishing Company, Reading, MA, 1995.

Tolvanen, J.-P., Incremental Method Engineering with Modeling Tools. Ph.D. Thesis, Jyväskylä Studies in Computer Science, Economics and Statistics, University of Jyväskylä, No. 47, 1998.

de Vasconcelos, F., Werner, C., Organizing the software Development process Knowledge: An Approach Based on Patterns. International Journal of Software Engineering and Knowledge Engineering, Vol. 8, no. 4, 1998, pp. 461-482.

Vidgen, R., Braa, K., Balancing Interpretation and Intervention in Information System Research: The Action Case Approach. Lee, A., Liebenau, J., DeGross, J. (eds.), Information Systems and Qualitative Research. IFIP, Chapman & Hall, London, 1997, pp. 524-541.

Wartik, S., Prieto-Díaz, R., Criteria for Comparing Reuse-oriented Domain Analysis Approaches. International Journal of Software Engineering and Knowledge Engineering, Vol. 2, No. 3, September 1992, pp. 403-431.

Wieringa, R., A Survey of Structured and Object-Oriented Software Specification Methods and Techniques. ACM Computing Surveys, Vol. 30, No. 4, December 1998, pp. 459-527.

Yates, J., Orlikowski, W.J., Genres of Organizational Communication: A Structurational Approach to Studying Communication and Media, Academy of Management Review, Vol. 17, No. 2, 1992, pp. 299-326.

Yourdon, E., (ed.) Classics in software Engineering. Yourdon Press, New York, NY, 1979.

Yourdon, E., Modern Structured Analysis. Prentice Hall, Englewood Cliffs, NJ, 1989.

2 EVALUATION OF COMPONENT-BASED SOFTWARE DEVELOPMENT METHODOLOGIES

Forsell, M.[†], Halttunen, V.[†], Ahonen, J.[†], "Evaluation of Component-Based Software Development Methodologies". Penjam, J. (ed.), Proceedings of the Fenno-Ugric Symposium on Software Tecnology, FUSST'99, Tallin, Estonia, Institute of Cybernetics at Tallinn Technical University, 1999, pp. 53-63.

©Institute of Cybernetics at TTU, Tallinn, Estonia, 1999. Reprinted with permission.

[†]Information Technology Research Institute, University of Jyväskylä,

Abstract

There are hopes that by using components the quality of software products can be improved. Besides, software production could become faster and the complexity of programs would be more controllable. Component-based software development aims at reusable computer programs. In order to harvest all the potential benefits from component-based software development, more emphasis should be put on the systematicity of the software development process: there is a need for valid methodologies that guide the software development process. In this study we evaluate three methodologies that support component-based thinking. In the evaluation we utilized the NIMSAD framework. Through evaluation we try to expose the similarities and differences in the evaluated methodologies. The similarities show us, we believe, the areas where a certain level of a consensus has been reached, while the differences not only tell us about missing consensus but can also reveal weak points in component software development seen as an entire process.

2.1 Introduction

In everyday life the importance of software programs is rapidly increasing. Household appliances, entertainment electronics and Internet services are examples of areas where ever more computer programs are needed. At the same time, the nature of working life has become more information intensive: in brief, we have entered into the information society. The users of different software are more aware of the opportunities of modern information technologies and require constantly new features in the software products. As a result of the increasing size and usage of programs more bugs can be found in software [10]. So, how to keep the quality of software high?

Taivalsaari [23] has argued that the software business will continue on the current basis as long as the profits remain high. Today, it seems that it is not a question of money: the need for improved software development processes is inevitable because of the enormous need for new software products. One piece of evidence for this trend is that in Europe and USA there is a shortage of 700 000 people in the software industry [20]. Besides increasing education of professionals in software development we also need new technologies and methods to develop high quality software products at an increasing speed. One solution could be the reuse of programs or program components [18], [15], [16].

Reusability technologies can be divided into two major categories: composition technologies and generation technologies [3]. Composition-based technologies aim to build programs from atomic components, whereas generation technologies aim to generate programs from recurring patterns or structures (i.e., 4 GL). In this study we concentrate on composition-based approaches. Component reuse at the moment is more or less ad hoc and it is usually based on one developer's knowledge and skills [5]. Computer programs are developed in almost every house, and, in relation to the extent of the software business, there are too few professional features in the processes of software production. As Jackson [13] put it, one can call himself a software engineer if he has bought a visual programming tool (like Visual Basic) and has learnt to make 'programs' with it.

Getting closer to the computer's 'soul', some successful examples of code reuse can be found: UNIX subprograms, mathematical and program language libraries, database interfaces and GUI toolkits. All of these are limited to a narrow domain which is well understood. Software developers can quickly learn these domains through education and work [3]. Research into components has also concentrated on a limited view: the focus has been on the interfaces and properties of components rather than on the entire process of component creation and use. Furthermore, there is no integrated model of component use in software development [7]. Reuse is not a thing that just happens. Basili [2]stated in 1994 that current software development methodologies more or less prevent reuse. Component usage has to be planned [11], [17], [22], and therefore we need methodologies which are seen as a means to achieve a more rigid approach to software development (cf.[1]). In component-based software development methodologies (CBM) we have to bring quality to component use and documentation. CBMs have to pay attention to the context where they are used because, without analyzing the context, it is problematic to define what a component is and whether or not components can be successfully utilized [5].

Evaluation of different software development methodologies has been carried out (i.e., object-oriented methodologies [9]) but so far there is no comparative studies on component-based methodologies. In this study we aim to contribute to this issue. Our objective is (1) to find similarities in the methodologies, through which we attempt to expose the common features and aspects in component-based software development, and (2) to analyze differences between methodologies to point out areas where consensus has not been achieved For the possible methodologies we set the following two criteria: (1) the component usage viewpoint had to be explicitly taken into account, and (2) the methodology should have been published in sufficient breadth (i.e. as a book). Using these criteria we selected the following methodologies for deeper consideration: Catalysis [6], OMT++[10], and Unified Process [12]. Each of these methodologies uses components in their process and is described in a recently published book (in 1999).

2.2 Evaluation framework

Different approaches to comparing methodologies are considered by Avison and Fitzgerald [1], and Nielsen [19], for example. Based on these two studies we chose the NIMSAD[1] framework [14] as our analysis tool. This framework was chosen for the following reasons: (1) it has a wide scope (it uses the entire problem solving process as the basis of evaluation), (2) it is not restricted to evaluation of any particular category of methodologies, (3) it is practical, it has been used in several real-life cases (cf. [19], [14]), and (4) it considers different use situations (cf. [19]). According to NIMSAD, methodologies are evaluated through four elements, which are: the methodology context, the methodology user, the methodology, and finally the way the methodology evaluates the other three elements (see Table 1).

The methodology context means the situation where the methodology is intended to be used and what is considered as important in this situation. Usually, the context is the organization whose problems are to be solved by the software being developed. The software development process is supported by the methodology which, in turn, is used by the methodology user. So, the methodology user is the developer. It is necessary to know what guides his decisions, what kind of abstract thinking is required from him, how well he has to know the methodology he utilizes, and how he can acquire the necessary skills. These things are included in the evaluation of the methodology user element.

From the methodology element we would like to know how it supports the problem-solving process. A methodology guides and assists the methodology user in seeing the problem in certain way, in asking relevant questions, and in

[1] Normative Information Model-based Systems Analysis and Design

TABLE 2.1 The four elements of the NIMSAD framework and the questions used in the analysis.

Elements and Questions	Explanation
Methodology Context (MC)	
Use situation	What kind of situations does the methodology suit?
Start for methodology use	Which incidents initiate the use of the methodology?
Customers and problem owners	Who are the customers and problem owners (apart from users)?
Context description	How is the context described for the system to be built?
Culture and politics of methodology use	What is the culture and politics of methodology use, explicitly and implicitly (in parenthesis)?
Risks in describing context	What risks does the methodology identify when describing context?
Risks of methodology	What are the risks in using the methodology?
Methodology User (MU)	
Users motives and values	What are users' motives and values (implicitly)?
Needed abstract reasoning	What level of abstract reasoning is required from the user?
Needed skills	What skills does the user need to accomplish tasks required in methodology use?
Methodology (M)	
Problem situation and problem boundaries	How does the methodology help in understanding the particular situation and the boundary setting?
Diagnosis of situation	How do you diagnose what kind of system is needed?
Prognoses for system	How do you make a prognosis for the system to be built?
Problem defining	How do you define problems which need to be solved?
Deriving notional systems	How you get systems which need to be described?
Logical design	Is this phase done? How do you implement this phase?
Physical design	Is this phase done? How do you implement this phase?
Implementing the designs	Is the implementation phase described? What is included in it?
Evaluation (E)	
Before intervention	How are MC/MU/M evaluated before intervention?
During intervention	How are MC/MU/M evaluated during intervention?
After intervention	How are MC/MU/M evaluated after intervention?

overcoming any new problems that arise. Thus, regarding the methodology element, we evaluate how the methodology supports the problem-solving process, that is: problem formulation, solution design, and implementation. In Table 2.1 the three basic stages are further distinguished as follows: Problem formulation consists of understanding the situation, performing the diagnosis, defining the prognosis outline, defining problems, deriving notional systems. Solution design consists of performing conceptual/logical design, performing physical design. Implementation is also one stage in Table 2.1 (implementing the designs).

Jayaratna [14] defines an extensive set of questions to analyze each element of the NIMSAD framework. Avison and Fitzgerald [1] have summed up Jayaratna's questions. We have used their list of questions as a starting point in our analysis.

Jayaratna [14] emphasizes that NIMSAD does not try to rank methodologies. We would also like to remind the reader that we are not going to reveal one methodology's superiority or inferiority but our purpose is to analyze whether some of the evaluated elements are taken into account in the methodology or not. When selecting methodologies for practical purpose, the evaluator should set his own criteria for evaluating methodologies (based on the context where the methodology is used) but he can use our analysis as a starting point or a frame for a more detailed evaluation that pursues the selection of the 'best methodology'. NIMSAD especially stresses the early phases of methodology use, which are problem formulation and problem situation. Also the methodology context and the methodology's 'internal' evaluation (self-evaluation) are seen as crucial in the framework. We underline the role of the evaluation element by citing Jayaratna's own words in [14] on page 108:

> "No problem-solving process can be considered complete until evaluation has been carried out. It is the evaluation which helps us to measure the effectiveness of the problem-solving process and the problem solver in the 'problem situation' - unless this element is considered there is no way of establishing that the 'problems' have been successfully resolved. Despite the importance of evaluation, however, we find that very few methodologies incorporate it into their steps."

2.3 Evaluation

In this section we present the results of our study where Catalysis, OMT++ and Unified Process were evaluated. The element methodology is considered a problem-solving process comprising the three major phases introduced in Section 2: problem formulation, solution design and implementation. The summary of results is presented in Table 2.2.

TABLE 2.2 Results of the evaluation of component-based methodologies according to NIMSAD framework elements and selected viewpoints of each element.

Element	Catalysis	OMT++	Unified Process
Methodology Context			
Use situation	All	Large, interactive systems	All
Start for methodology use	User requirements, BPR notification	New features, earlier plans, customer needs	Former Systems, earlier plans, User requirements
Customers and problem owners	System users, customer management	System users, system developers	Mass markets, organizations, in-house
Context description	Business models	Domain models, Use Cases	Business models, domain models
Culture and politics of methodology use	N/A (technical rationality)	N/A, (technical rationality)	N/A (technical rationality)
Risks in describing context	Customer does not know what he wants	N/A, assumes that qualified people do this	Too precise descriptions
Risks of methodology use in describing context	Customers are seen as problems, assumes correct requirements	Inexperienced methodology users, assumes correct requirements	Relays methodology user's abilities/knowledge
Methodology User			
User motives, values	N/A (technical rationality)	N/A (technical rationality)	N/A (technical rationality)
Needed abstract reasoning	High, new concepts for CBD are introduced	Early in life cycle needed more	High, early in life cycle needed more
Needed skills	Domain and methodology knowledge	Domain, methodology, programming language	Methodology, Domain
Methodology			
Problem situation and problem boundaries	Use Case models, customer feedback, existing information, dictionary	Requirements, Use Case models, former plans, customer reports, dictionary	Business and domain models, dictionary, former systems, customers

continues

TABLE 2.2 *(continued)*

Element	Catalysis	OMT++	Unified Process
Diagnosis of situation	Use Cases, Type models, System Context models	Domain models, use cases, requirements lists	Business and domain models, Use Cases, requirements lists
Prognoses for system	N/A	Use Cases	Vision of the best possible system
Problem defining	N/A	Requirements classification and evaluation	Technical problems
Deriving notional systems	N/A	N/A (Use cases)	N/A (Use Cases, software architecture)
Logical design	Boundaries between humans and between components	Conceptual models, solutions to the problem, User interface	Architecture description, Use cases, use case analysis, analysis models
Physical design	Internals of components	MVC++ -model	Use case designs, design models
Implementing the designs	N/A, some strategies for iterative approach	Programming, testing, debugging	Programming, testing, transition to customer
Evaluation			
Before intervention	N/A	N/A	N/A
During intervention	N/A	N/A	N/A
After intervention	N/A	N/A	Methodology User, Methodology

2.3.1 Element 1: Methodology context

Context description is seen as crucial in every methodology. Unified Process seemed to be most in-depth in context description. All the three evaluated methodologies assume that situation, problem and context can be described in to the extent that development work can continue based on the information gathered, although the coverage or quality of the information is not formally tested. Catalysis comes closest to formal testing in formally describing pre- and post-conditions and in stressing formalism in descriptions.

Catalysis and Unified Process regard the methodology as suitable in almost every possible software development situation. Unified Process presents itself as an umbrella for software development processes and it can be applied to diversified applications areas, different types of organizations, different competence levels, and different project sizes [12]. Similarly, according to Catalysis, it suits embedded software development as well as regular business software development. Catalysis provides a set of process patterns from which one can select those appropriate for different situations. OMT++ considers itself suitable for large interactive software development. Altogether, consensus prevails in that all the evaluated methodologies believed that every methodology has always to be suited to the particular development situation.

The methodologies consider the methodology context an important factor, and they recommend domain modeling for describing the context where the methodology is used. By modeling the context we can clarify the function of the domain, and we can see what skills and what kind of knowledge are required within the domain. The methodologies employ ′use cases′ for modeling the problem and defining the system being developed. In our estimation, Unified Process provides most depth to modeling the context.

As a result of this study we believe that Unified Process may serve the developer as an umbrella for different approaches of software development. Instead, we were not convinced of the suitability of Catalysis for this purpose. OMT++ is claimed to suit large interactive systems development (especially at Nokia). On the basis of our study we have no argument against it.

2.3.2 Element 2: Methodology User

The methodology user′s values and motives are implicitly presented in methodologies but all the evaluated methodologies assume that the methodology user is driven by technical rationality so that it is both his and the customer′s aim to produce the rationally best system on the basis of the customer′s needs.

According to the evaluated methodologies the methodology user has to be experienced and he has to know the domain well. By an ′experienced user′ the methodologies imply a user who has been working in the software development field for several years and knows how the business (domain) works. Furthermore, he has to know the stages of the methodology and the used notations. Experience was considered so important that identifying, defining and modeling components was left only to experienced developers. Experience was not seen as equally critical in the design and implementation phases. Only Catalysis uses design patterns and frameworks as an important part of the methodology. By using these it aims to help ″structuring″ the methodology users′ experience in such a way that the solutions arrived at once can be reused later.

2.3.3 Element 3: Methodology

Problem formulation. With regard to problem formulation, the evaluated methodologies see customers merely as a means to obtain feedback and necessary information. Their participation in problem formulation is only marginal.

However, we have to keep in mind that problem formulation is embedded or implicitly assumed to be part of the gathering requirements for the system. During problem formulation methodologies set limits on the problem domain by using domain models and use cases. These descriptions depict which parts of the problem domain are included in and which are excluded from the system.

Problem formulation is not seen as fruitful for component use. Nearest to this is Catalysis which describes the problem domain with type models which themselves can include patterns and frameworks and which later on are used as a basis for component identification and use.

Solution Design

In Catalysis it is crucial to find types which help fulfil requirements set for the software. Types are seen as a set of objects that conform to a given type specification.

OMT++ relies on its own MVC++ design pattern when designing a solution. In the logical/conceptual design phase it is, according to OMT++, crucial to find the central concepts of the problem domain and to design the solution according to these found concepts. The resulting model is called an analysis model. Also, in the logical/conceptual design phase, the user interface is designed, and it has to be accepted by the customer. In the physical design phase the analysis design model and interface model are linked according to the MVC++ design pattern.

For Unified Process it is essential to create a software architecture that works. In the logical/conceptual phase the software architecture is designed so far that one can be sure that also the forthcoming parts of the software and the changes to the software are taken into account in the architecture. In Unified Process the software development process is guided by use cases and the software architecture. Unified Process and OMT++ see components as code components, the main purpose of which is to replace existing components. Catalysis puts more emphasis on design patterns and frameworks in problem solution design. Any particular interest in finding and reusing components made once, was not found in the evaluated methodologies. Catalysis takes note of the reuse of components made earlier but only to a minor extent. More weight was given to defining and implementing new components.

Design Implementation

OMT++ gives most guidance for implementation. In OMT++, for example, mapping the object model to the rational database is explained in depth.

Unified Process gives the most extensive description of how to hand over the software system to the customers and does this in two phases. First, the beta version of the software is released to selected users. After testing and correction of the system it is delivered in its entirety to the customer. During beta version testing experience of use is collected, possible errors are corrected, educational material is created, setup programs and manuals are finalized, etc.

Catalysis does not take note of design implementation to any noticeable extent. It appears that the evaluated methodologies see language libraries, GUI-kits, databases and so forth business as normal in the implementation phase because use of them is not discussed with any particular meaning.

2.3.4 Element 4: Evaluation

Only Unified Process points out the importance of evaluating the use of the methodology. It also gives instructions to do this but the evaluation is carried out after the utilization of the methodology (only) and it concerns the methodology user and the methodology itself after each phase of the process.

2.3.5 Summary of Evaluation

Generally speaking the evaluated methodologies have many features that are similar. This can be explained by the fact that all of them are object-oriented and use Unified Modeling Language (UML). There are no major differences in the way they model components. This can imply that UML is seen as an adequate means to model systems so that the user of software components can get sufficient information. The developer's experience seems to be crucial in component use and definition. Also the methodology user's competence is crucial in deriving requirements from the interactions with the customer.

The differences between the methodologies lie in the development principles. Catalysis emphasizes types and type models which are also a central means to define and model components. Catalysis also puts weight on formalism, so that the models at different levels of abstraction use the same concepts. OMT++ seems to be a very practical methodology, which provides a smooth transition from analysis to design. OMT++ builds upon its own MVC++ design pattern, which is a central part of the methodology. Unified Process, instead, places particular stress on 'use cases' which guide the methodology users throughout the process. By employing use cases it is possible to collect requirements, to analyze and design the system, and finally to test the system. Another crucial aspect for Unified Process is the software architecture, which is created by the most influential use cases as early as possible in the development cycle.

2.4 Implications and further research

This study evaluated methodologies at quite a general level. It does not explicitly reveal how well a methodology supports component-based software development. The study, however, contributes to the understanding of the current situation of component-based methodologies, and especially it reveals a few areas where these and other methodologies could be improved. In the following we analyze the implications of our study with regard to both the evaluation framework (NIMSAD) and the evaluated methodologies.

The NIMSAD framework does not give any criteria for finding the 'best' methodology. The companies pursuing this kind of selection should create their own criteria, based on their needs and the situation where the methodology is to be applied. Nevertheless, the following general conclusions about the evaluated methodologies can be drawn:

- OMT++ gives quite a consistent and clear way of transforming logical models to physical ones so that the physical models are easy to implement.

- Catalysis emphasizes a formal approach and describes how pre- and post-conditions for components can be created. Through these conditions one can describe software requirements as specifically as needed.

- Unified Process puts much weight on the beginning and the end of the development cycle; that is, on the one hand, on describing the context and, on the other, on the phase where the software is passed to the customer.

From the component viewpoint, it is quite disturbing that using components is left solely to the experienced developer. What if we do not have suitable resources? We suggest that information about components is added in such a form that even less experienced developers can use the components. Another alternative is that experienced developers' experiences are recorded in design patterns as described in Gamma et. al. [8]. Evaluation is almost totally neglected by methodologies, only Unified Process carries out evaluation and this is done only at the end of the project. Methodologies should put more weight on evaluation so that even before and during methodology use one could evaluate the methodology and the methodology user. In our opinion, evaluating the context might make methodology use too cumbersome. Unified Process and Catalysis see themselves as general development process frames. We are waiting for experiences and examples of how to specialize these methodologies in different contexts. Problem formulation is implicit in all the evaluated methodologies We would not see it as a bad thing if this was an explicit phase in methodologies. Finally, implementation, at least component implementation, is presented quite briefly. Perhaps object-oriented programming from models which are generated during development is so straightforward that these questions are unimportant but we consider that guidelines or templates for component implementation would be important.

NIMSAD is quite exacting as a evaluation framework and the use of it requires much time. It is not that straightforward to answer all of the elements included and part of the questions presented in the book overlap and are ambiguous. First of all, when starting to use the NIMSAD framework, it is good to start by defining the questions or viewpoints you want to answer concerning each element. Many questions presented in NIMSAD methodologies are answered only implicitly. Therefore, ready-made specific classifications are sometimes needed. For example, when defining how culture and politics affect the methodology context, you can use Schön's [21] classification technical rationality and reflection-in-action. At the moment, after using NIMSAD, the evaluator has to create specific criteria and re-evaluate every methodology according to specific criteria and, moreover, if any criteria are outside NIMSAD there is no means to take these criteria into account.

We see the following features of the NIMSAD as confusing: (1) the methodology user is separated from the methodology context; we see that the methodology user becomes part of the methodology context at the moment he starts to use the methodology, (2) Jayaratna's viewpoint on what epistemology and ontology are differs from commonly accepted viewpoints (compare Jayaratna's [14] comments on pages 40-41 to, for example, Burrell and Morgan's [4] comments on ontology and epistemology), and (3) some concepts are used ambiguously; for example on pages 30-31 Jayaratna [14] writes about the waterfall model but in the clarifying picture it is called the waterfall method.

From the above we can derive three essential research topics: (1) to further study which components are explicitly taken into account by the methodologies, (2) to improve methodologies in making a context description (in the component view sense), and to improve the component use process in methodologies, and (3) to create a way of adding evaluation criteria to NIMSAD in such a way that it supports the selection of methodology in a particular use and context.

2.5 Conclusions

We evaluated three component-based methodologies according to the NIMSAD framework [14]. The evaluated methodologies, Catalysis [6], OMT++[10] and Unified Process [12], are quite similar in the sense that they all are object-oriented, they use UML as a description language, they see software architecture as a driving force in component-based development, and they see that experienced developers are mandatory if component-based development is to succeed. The methodologies differ from each other with regard to (1) the phases of a development process when components are used, and (2) the types of components used.

Acknowledgements

This research was supported by Tekes Technology Development Centre, Finland, and companies participating in the PISKO project.

References

1. Avison, D., Fitzgerald, G.: Information Systems Development: Methodologies, Techniques and Tools, 2nd Edition. McGraw-Hill International (UK) Limited (1995)

2. Basili, V.: Facts and myths affecting software reuse. Proceedings of the 16th International Conference on Software Engineering (1994), 269

3. Biggerstaff, T., Richter, C.: Reusability framework, assessment, and directions. IEEE Software, March (1987), 41-49

4. Burrell, G., Morgan, G.: Sociological Paradigms and Organisational Analysis. Gower Publishing Company Ltd. (1989)

5. Caldiera, G., Basili V.: Identifying and Qualifying Reusable Software Components. IEEE Computer, February (1991), 61-70

6. D'Souza, D., Wills, A.: Objects, Components, and Frameworks with UML: The Catalysis Approach. Addison-Wesley (1999)

7. Dusink, L, van Katwijk, J.: Reuse dimensions. Proceedings of the 17th International Conference on Software Engineering on Symposium on Software Reusability (1995), 137-149

8. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)

9. Hutt, A. (editor): Object Analysis and Design: Comparison of Methods. John Wiley & Sons, Inc. (1994)

10. Jaaksi, A., Aalto, J-M., Aalto, A., Vättö, K.: Tried & True Object Development: Industry-Proven Approaches with UML. Cambridge University Press (1999)

11. Jacobson, I., Griss, M., Jonsson, P.: Software Reuse: Architecture, Process and Organization for Business Success. ACM Press (1997)

12. Jacobson, I., Booch, G., Rumbaugh, J.: The Unified Software Development Process. Addison-Wesley (1999)

13. Jackson, M.: Problems, methods and specialization. Software Engineering Journal, November (1994), 249-255. Also available at ftp://st.cs.uiuc.edu/pub/patterns/papers/problem-frames.ps (21.5.1999)

14. Jayaratna, N.: Understanding and Evaluating Methodologies. McGraw-Hill Book Company (1994)

15. Krueger, C.: Software reuse. ACM Computing Surveys, Vol. 24, No. 2, June (1992), 131-183

16. Liao H-C., Chen M-F., Wang, F-J.: A domain-independent software reuse framework based on a hierarchical thesaurus. Software - Practice and Experience, Vol. 28, No. 8, July (1998), 799-818

17. Lim, W.: Managing Software Reuse. Prentice Hall PTR (1998)

18. McIlroy, M.: Mass produced software components. In Naur, P., Randell, B. (editors): Software Engineering; report on a conference by the NATO Science Committee (Garmisch, Germany), October (1968), 138-150

19. Nielsen, P.: Approaches to appreciate information systems methodologies: a soft systems survey. Scandinavian Journal of Information Systems, Vol. 2, (1990), 43-60

20. Nukari, J., Forsell, M.: Finnish Software Industry's Growth Strategy and Challenges. Teknologiakatsaus 67/99, TEKES, Paino-Center Oy, Sipoo (1999) (In Finnish)

21. Schön, D.: Educating the Reflective Practitioner. Jossey-Bass Inc. (1987), 78-79

22. Szyperski, C.: Component Software. Addison-Wesley (1997)

23. Taivalsaari, A.: A Critical View of Inheritance and Reusability in Object-oriented Programming. Ph.D. Dissertation, Jyväskylä Studies in Computer Science, Economics and Statistics, No. 23, University of Jyväskylä (1993)

24. Tolvanen, J-P.: Incremental Method Engineering with Modeling Tools: Theoretical Principles and Empirical Evidence. Ph.D. Dissertation, Jyväskylä Studies in Computer Science, Economics and Statistics, No. 47, University of Jyväskylä (1998)

3 USE AND IDENTIFICATION OF COMPONENTS IN COMPONENT-BASED SOFTWARE DEVELOPMENT METHODS

Forsell, M.[†], Halttunen, V.[†], Ahonen, J.[†], "Use and Identification of Components in Component-Based Software Development Methods". Frakes, W. (ed.), Software Reuse: Advances in Software Reusablitiy. Proceedings of the 6th International Conference, ICSR-6, Vienna, Austria, June 2000, Lecture Notes in Computer Science 1844, Springer-Verlag, Berlin, Heidelberg, 2000, pp. 284-301.
©Springer-Verlag, 2000. Reprinted with permission.

[†]Information Technology Research Institute, University of Jyväskylä, Finland.

Abstract

New software systems are needed ever more but to keep up with this trend software developers must learn to create quality software more efficiently. One approach is to (re-)use components as building blocks of the new software. Recently there has been more interest to create component-based software development methods to support this. In this article we first set out requirements for reuse-based software development and then evaluate three component-based methods, namely Catalysis, OMT++, and Unified Process. As a conclusion we argue that evaluated methods produce prefabricated components and that component-based means that software developers can change better components to existing systems. Reuse of components to create new software is neglected in these methods.

3.1 Introduction

While the users' requirements and needs for software products are rapidly increasing, the number of software professionals is not accompanying this trend.

Every time new software system is created at least half of the developers are needed to keep it going on. This is due maintenance needs. One solution to produce quality software systems more efficiently is to use components as building blocks of the new software. This fact means that software components need to be more reusable and reused.

The very basic idea of reuse is: use a thing more than once [3, 32, 20]. A component is the common term for a re-usable piece of software. Depending on the level of abstraction and the ways of selection, specialization to the specific situation and integration to the whole, reuse technologies can be categorized as follows [20]:

1. high-level languages
2. design and code scavenging
3. source code components
4. software schemas
5. application generators
6. very high-level languages
7. transformational systems
8. software architectures

A more coarse division distinguishes two kinds of reuse technologies: composition and generation. In composition technologies the software is *composed* of its parts (components) whereas in generation technologies the software is generated from the higher-level descriptions or specifications.

Successful areas of software reuse, such as UNIX subroutines, mathematical or programming language specific program libraries, database management systems, or tools for graphical user interface, are well known and carefully explored and these areas have well established patterns [5]. A software developer can easily learn the use of them through education and working experience.

Reuse does not just happen it requires careful planning and coordination [23, 17, 13]. However, the research of software reuse usually concentrates on the properties of software component or their interfaces without having a proper understanding on the whole *process of software reuse*. Dusink and Katwijk [12] note that there is no integrated model available for reuse support in the software development. Reuse-based software development requires specific methods. However, good "reuse methods" are not available. As Basili [2] puts it, current software development methods even hinder reuse in system development. Furthermore, Bailey and Basili [1] argue that the reuse models, which build upon expertise of the application domain without having clear instructions on how to do things, offer insufficient guidelines for the reuse process. In brief, such models burden the experts too much and make, thus, the process vulnerable.

The objective of this paper is to assess how current methods support reuse in software development, and, through the evaluation, to expose how to improve methods aiming at software reuse. For the possible methods we set the following two criteria:

1. component usage viewpoint had to be explicitly taken into account, and
2. the method should have been published in a sufficient breadth (i.e. as a book).

By these criteria we selected the following methods for deeper consideration: Catalysis [11], OMT++ [16], and Unified Process [18]. Each of these methods uses components in their process and is described in a lately published book (in 1999).

In this paper we approach software reuse from the above mentioned composition aspect. We consider 'a component' in a wider sense than traditionally. Thus, a component is not restricted to be a code component, but it can be also some other artifact like software schemas or software architectures (see [20]). We see also many other reusable elements in software development, for example, domain models, analysis and design documents etc.

When analyzing the support of a method for the reuse process, it is necessary to define the process model, through which the analysis is done. For finding such a model we first scan for the essential features of a reuse process and then — according to these features — select a process model that can be considered "ideal".

3.2 Crucial Features of a Reuse Process

Traditional software development processes require substantial consideration when component-based reuse is adopted. In this part of the paper we make a literature review to find out the most crucial features of a reuse process gained attention of the contributors of the area. We have listed the features that have been noticed by several contributors. These are:

1. Domain modeling [9, 26, 24, 6, 10, 15, 8, 4]
2. Software architecture [17, 25, 8, 20]
3. "More than code" [15, 7, 21, 22, 19, 13]
4. Separate processes for component (re)use and creation [7, 30, 19, 32, 23, 5, 17]
5. (Component) repository [20, 28, 27, 14, 26, 10]

There is a wide consensus that domain modeling and software architecture are pre-requisites for successful use of software components. It is absolutely necessary to know the context where the components are (re-)used, because it is not self-evident how well the components suite to different contexts (see [7]). The identification of components starts with domain analysis, which defines the reusable components at the conceptual level. Therefore, domain analysis is an important phase in finding out what components are needed and how they interrelate. Software architecture, which defines how the software is composed of its subsystems, helps to manage with the software as a whole, which makes easier the selection of suitable components. The domain analysis and the software architecture form together a sound basis for use of components in the particular application domain (see [10] and [26]).

Despite the fact that reuse has traditionally meant reuse of code, it is necessary to realize that reuse can — and should — be extended to many other artifacts, too. Like Horowitz and Munson [15] state, reuse can happen at several levels and the best benefits can be achieved when, in addition to the reuse of code, requirement analysis, designs, testing plans etc. are reused. Caldiera and Basili [7] argue that experience is also an important reusable resource. Reuse of experience enables other types of reuse. Lanergan and Grasso [21] report highly successful reuse of the software's logic structures.

It is the size and the abstraction level of the components that decides the phase of the development process wherefrom the components can be used [22]. When talking about reuse of components, two different processes can be separated: production of components and use of them [32, 30, 5]. In addition to this, some researchers see the maintenance of the component repository as a separate task [23, 27, 10, 7, 26]. Some emphasize the role of managing the reuse infrastructure (e.g. [17] and [23]). Moreover, some researchers have considered abstraction as a means to find a component (see [26] and [19]).

The most extensive models for reuse include the STARS [31] and the models by Lim [23] and Karlsson [19]. The Karlsson's model considers reuse purely from the object-oriented point of view. The coverage of the STARS model and the Lim's model is very similar. However, it seems that the STARS model is a little more directed towards code-components. The Lim's model is not dependent on any software development paradigm. It notifies the essential features of a reuse process and thus it sees components as "more than code". We selected the Lim's model for our evaluation.

3.3 The Lim's Model of the Reuse Process

Lim [23] talks about 'assets' instead of 'components'. This shifts the emphasis from pure code orientation to conceiving 'a component' as a wide concept that covers designs and other things besides computer programs. We believe that perspective is absolutely necessary when aiming at successful and wide-spread reuse of components in software development.

Lim divides the reuse process into the following four major activities:

1. managing the reuse infrastructure
2. producing reusable assets
3. brokering reusable assets
4. consuming reusable assets

First activity, managing the reuse infrastructure, plans and drives the other three activities. It manages the whole reuse process. Lim's [23] subtasks of the three latter activities are described more closely in Table 3.1. Lim names distinct roles for the developers in three latter activities. *Producers* create assets for reuse, *brokers* provide repository and support for reusable assets, and *consumers* produce new software with reusable assets.

Domain analysis and software architecture are considered in phases analyzing domain and producing assets respectively. In Lim's model it is very obvious that target of the reuse is much more than merely the code and that for producing and using of the components are two separate processes. Repository and use of it are in very central position in Lim's reuse model. To be successful reuse has to be planned carefully.

TABLE 3.1 The reuse process, its phases and explanations.

Activity / Tasks	Explanation
Producing Reusable Assets (PRA)	Reusable assets can be created either by prefabrication or retrofitting. In either case, the production of the reusable asset is preceded by domain analysis (DA). Two major elements of PRA are domain analysis and domain engineering.
Analyzing Domain	Tasks: • DA attempts to create a domain model which generalizes all systems within a domain. • DA is at a higher level of abstraction than systems analysis. • The resulting assets from a DA possess the functionality necessary for applications developed in that domain.
	Continues on next page

Continued from previous page (Reuse process)

Activity / Tasks	Explanation
Producing Assets	Tasks: • Produced assets are those identified by the DA and include both components and architectures. • The architecture is the structure and relationship among the constituent components. Having identified the set of assets that have a high number of future reuse instances. • Two approaches are available for producing these assets: prefabrication and retrofitting. *Prefabricating* is approach to "build" reusability into assets when they are created. This approach has been variously called "design for reuse" and "a posteriori reuse". *Retrofitting* is second approach to examine existing assets, evaluate the feasibility of reengineering them for reuse, and if viable, doing so. This set of activities has been called salvaging, scavenging, mining, leveraging, or a priori reuse.
Maintaining & Enhancing Assets	Tasks: • Maintenance involves changing the software system / product after it has been delivered. Maintenance can be • Perfective maintenance (enhancing the performance or other attributes), • Corrective maintenance (fixing defects), or • Adaptive maintenance (accommodating a changed environment). • When assets are enhanced, fixed, or replaced, the consumers are notified of the changes, and in many cases for active projects, will require integration of the newer versions of the assets. • Verification and validation (V&V) are performed throughout the life cycle. In software reuse, V&V are intended to demonstrate that the reusable asset will perform without fault under its intended conditions.
Brokering Reusable Assets (BRA)	Aids the reuse effort by qualifying or certifying, configuring, maintaining, and promoting reusable assets. It also involves classifying and retrieving assets in the reuse library.
Assessing Assets	Tasks: • Potential assets from both external and internal sources should be assessed before order. • Potential assets are identified and brokers examine them, reviewing several factors.
Procuring Assets	Broker determines whether to purchase or license the asset, purchase and reengineer the asset to match the consumers' needs, produce the asset in-house, or reengineer an existing in-house non reusable asset to meet consumer needs.

Continues on next page

Activity / Tasks	*Continued from previous page (Reuse process)* Explanation
Certifying Assets	Tasks: • Reusable assets should be certified before they are accepted into the repository. • Certification involves examining an asset to ensure that it fulfills requirements, meets quality levels, and is accompanied by the necessary information. Once the asset is certified, the next step in the process is to accept the asset.
Adding Assets	Adding an asset involves formally cataloging, classifying, describing it, and finally entering it to the list of reusable assets.
Deleting Assets	The broker should examine the inventory of assets and delete those which are not worth continuing to carry or have been superseded by other reusable assets.
Consuming Reusable Assets (CRA)	CRA involves using these assets to create systems and products, or to modify existing systems and products. CRA is also known as *application engineering*.
Identifying System & Assets	Tasks: • End-users' needs are translated into system requirements. • Requirements for assets are also determined as part of this analysis. • In *reuse-enabled businesses*, system requirements are determined in part by the availability of reusable assets • In *strategy-driven reuse*, a deliberate decision is made to enter certain markets or product lines in order to economically and strategically optimize the creation and use of reusable assets which fulfill multiple system requirements.
Locating Assets	Consumers locate assets which meet or closely meet their requirements, using the reuse library, directory, or other means.
Assessing Assets for Consumption	Tasks: • Consumers evaluate the assets. • If a suitable asset cannot be found externally, the consumer must determine whether a reusable version should be requested from the producer group. In some circumstances, it may be more viable to create a non reusable version for the project at hand. • A modified asset may be valuable to other projects as well. Consequently, the consumer should consider submitting a request for a modification of the reusable asset which would be supported by the broker group.
Adapting / Modifying Assets	Asset is adapted to the particular development environment. If modification is necessary, the consumer should carefully document the changes. Possible reuse strategies are black box reuse and white box reuse.
Integrating / Incorporating Assets	Reusable assets are incorporated with new assets created for the application.

To cope with such a complicated process it is necessary to make difference between the above mentioned areas of the process. It is also important to know thoroughly the tasks of each phase.

3.4 Evaluation of Reuse Processes in Three Known Methods

The evaluation of the three methods, Catalysis, OMT++ and Unified Process, was started by reading carefully the following books: D'Souza and Wills [11], Jacobson et al. [18], and Jaaksi et al. [16]. Next, each method was evaluated in terms of its support for the activities and tasks of the Lim's model. In this section we summarize the main findings of our analysis. We start with a brief description each of the methods, after which we analyze the general features of them using the Lim's model.

3.4.1 Catalysis

Catalysis is based on three modeling concepts: type, collaboration, and refinement. Furthermore, it uses frameworks to describe recurring patterns. A *collaboration* defines a set of actions between group of objects. A *type* defines external behavior of an object. Precise description what is external behavior of the type is given in type model. Types serve as basic means to identify and document components. A *refinement* describes how abstract model maps to the more concrete ones. *Frameworks* are used to describe recurring patterns in specifications, models, and designs. [11]

Catalysis distinguishes between three levels of abstraction: domain/business level, component level, and internal design level. In the *domain level* one identifies the problem and defines the domain terminology and tries to understand the business processes. At the *component level* one specifies the system's boundary and distributes responsibilities among the identified/defined components. In the *internal design level* one implements specifications of the system and defines internal architecture, internal components and collaborations and designs the insides of the system and/or component. [11]

Finally, Catalysis is founded on three principles: abstraction, precision, and pluggable parts. In Catalysis *abstraction* means that one should focus on essential aspects one at the time while leaving others for later consideration. *Precision* means that one should be able to find out inconsistencies between specifications as early as possible, trace requirements through specifications or models, and allows to use support tools at a semantic level. *Pluggable parts* enables one to use results from the development work in the following projects. [11]

Catalysis does not give any specific process to produce software but it gives number of process patterns. By combining the process patterns one can create suitable process for current development needs.

3.4.2 OMT++

OMT++ uses OMT [29] as the backbone of the approach. The notations and naming conventions of OMT are used as is in OMT++. Even the methods name relies heavily on OMT.

OMT++ consists of four phases, namely object-oriented analysis, object-oriented design, object-oriented programming and testing. These phases are separate ones and they can be arranged either in a waterfall or iterative manner. In each of the phases there is activities that aim to model either the static or the functional properties of the system.

Although the use cases are seen as a very key to the successful development of software, for architectural and practical reasons the key abstractions are service blocks and components. A *service block* is a grouping of closely related components that provide a consistent set of reusable software assets to designers using the service block. A *component* is a configuration of files implementing a basic architectural building block, such as an executable program or a link library.

3.4.3 Unified Process

Rational Unified Process (RUP) is use-case driven, architecture-centric, iterative, and incremental. To serve its users the software system must correspond to user needs. RUP uses use cases to capture functional requirements which satisfies user needs. Additionally use cases drive the development process. Based on the found use cases developers of the software system create series of design and implementation models that realize the use cases. Thus *use-case driven* means that the development process follows a flow — it proceeds through a series of workflows that derive from the use cases.

By *architecture-centric* RUP means that a system's architecture is used as a primary artifact for conceptualizing, constructing, managing, and evolving the system under development. Full scale construction of the software system is not started before architecture designers can be sure that the developed architecture can manage through the software's lifecycle (i.e. maintenance and further development).

The *iterative and incremental* process in RUP means that the software system is developed in many iterations and through small increments. Each iteration deals with a group of use cases that together extend the usability of the product thus producing an increment to the whole software system.

Key abstractions are service packages, service subsystems, and components. *A service package* provides a set of services to its customers. Service packages and use cases are orthogonal concepts meaning that one use case is usually constructed by many service packages and one service package can be employed in several different use-case realizations. In RUP service packages are primary candidates for being reused, both within a system and across related systems.

TABLE 3.2 The scale for evaluating the features of the methods.

Symbol	Corresponding
-	Not mentioned at all or mentioned incidentally
+	Briefly considered
++	Distinctly considered
+++	Thoroughly considered

A *service subsystem* is based on the service package and there is usually one to one mapping between them. Usually service subsystem provide their services in terms of interfaces. Often service subsystems leads to a binary or executable component in the implementation. A *component* is the physical packaging of model elements, such as design classes in the design model. Stereotypes of components are for example executables, files, and libraries.

3.4.4 Evaluation of Methods

In the evaluation of the methods we analyzed the features of the methods against the Lim's model of a reuse process. The phases of "the ideal model" are depicted earlier in Table 3.1. We created a scale for estimating the support of the methods for each phase of the reuse process (Table 3.2). We remind that our evaluation is not based on experiences of using the methods but only on the book reviews we have accomplished.

The results of our analysis are summarized in Table 3.3. A more detailed analysis can be found in Appendix 1.

As Table 3 shows, the three evaluated methods have emphasis on producing components. In OMT++ and Unified Process this means production of *code* component whereas in Catalysis other types of components are also considered. The *use* of components has gained some attention, which, however, largely focuses on the identification of the system. The coverage of the methods is quite similar: they include domain modeling, production of components, and identification of a system. In practice, these elements seem to be intertwined. So, it is noteworthy that the methods, actually, integrates the production of components into the use of them.

According to the analyzed methods software production progresses as follows:

1. analyze the domain
2. identify the functionality of the system
3. define the software architecture
4. construct the software using components.

TABLE 3.3 Summary of the results.

Process / Phase	Catalysis	OMT++	Unified Process
Producing Reusable Assets (PRA)			
Analyzing domain	++	++	+++
Producing Assets			
Prefabricating	+++	+++	+++
Retrofitting	+	-	-
Asset Maintenance	-	-	-
Asset Enhancement	-	+	+
Brokering Reusable Assets (BRA)			
Assessing Assets	-	-	-
Procuring Assets	-	-	-
Certifying Assets	-	-	-
Adding Assets	+	-	-
Deleting Assets	-	-	-
Consuming Reusable Assets (CRA)			
Identifying System	+++	+++	+++
Identifying Assets	+	+	+
Locating Assets	-	-	-
Assessing Assets for Consumption	+	-	-
Adapting / Modifying Assets	+	+	+
Integrating / Incorporating Assets	-	-	-

The methods would be more usable if the production of components were distinctly separated from the use of components. Although components should be produced keeping the *reuse* aspect in mind, it is necessary to realize that the mentioned two tasks face different problems and different solutions and need to be managed as separate processes. For example, finding and adapting a component might get too little attention when these tasks are not seen as important processes of component reuse. This is due to the fact that the person responsible for producing a component naturally sees the component from the perspective of how the component is to be implemented, whereas the user of the component see the ″service″ provided by the component.

What is, then, the reason for integrating of the two tasks? Possible answers include:

- It is difficult to get acceptance for a method that suggests big changes into the prevailing practices.
- The developers of methods have not yet deep enough understanding on the information that is necessary when storing and retrieving components.
- If the producer and user of a component is the same person it may be difficult and even unnecessary to separate the two processes.

- Some methods use component purely for managing the complexity perhaps ignoring the reuse perspective.

The strength of the evaluated methods is their thoroughness in domain modeling and describing the software architecture. These two tasks, which

1. bind components to the context, and

2. define the connections between components,

are a crucial part of successful component-based software development. So, it seems that these areas are well covered by the current methods.

It appears that the telecommunication backgrounds of OMT++ and Unified Process have affected the development of these methods: since the telecommunication applications tend to be very complicated the methods, which are used to build such applications, must help splitting the application into parts that are manageable. The software architecture is in a crucial role when the maintenance of the application means replacing a component by a new one, or adding a new component to the old system.

We have already noted that the term 'component' has a wider meaning in Catalysis compared with the 'component' in OMT++ or in Unified Process. There is also another remarkable point where Catalysis differs from the other two methods. That is the way of building a component. Whereas OMT++ and Unified Process talk about service blocks or service packages, respectively, Catalysis derives components from the functions of the system. There is a big difference here, because these two approaches should be seen as orthogonal since one service package can be utilized by several functions. This implies that a function normally consists of several service packages or vice versa.

3.5 Discussion and Further Research

In the current methods the tasks of producing and using components are intertwined. This makes the methods complicated and decreases their usability. For example, storing and searching for a component can gain too little attention, when production of components dominates the software process. A concrete result of this problem is that no or little information on the components is saved for helping the further use of the components. It is obvious that in many cases the software people do not even know what is the necessary information that helps find a suitable component.

Are the contemporary component-based methods, then, really component-based? In our opinion, the answer is 'Yes' and 'No'. They are component-based in the sense that they aim at well-structured architectures, where the purpose of each element can be distinctly defined and where an element can be easily replaced by another element. They are component-based also in that they aim at well-defined interfaces. However, they have several weak areas that need to be improved if desired to fully benefit from the use of components. First, the methods should put much more emphasis on the sub-processes of the component use. These include identifying and searching for a component. The methods should have support to view the use of components from multi-purpose perspective. This means that the methods should not only help divide an application into its pieces but also find more generic features to be implemented in the component. Furthermore, the methods should include tools to collect relevant information on the components to be saved into a repository that can be effectively used when searching for a suitable component. The current methods have little or no support for using such a repository. Apparently Basili's [2] statement about hindrance of the current methods is still true.

It seems that the current practices in component-based software development still rely on software experts' personal knowledge. Although this point of view is understandable, it is, however, contradictory to the idea of using generic elements in software production. A real component orientation should, therefore, aim at practices and models that decrease the irreplaceableness of individual knowledge. This can be reached by standardization and other agreements but also by supporting the entire software process with information restored in repositories. Because the standardization process is usually a stony way, the importance of using repositories cannot be overestimated.

To direct the further research concerning the component-based methods we provide the themes that we see important:

- We should explore how to document components of different levels so that people not being expert of the domain could use them.
- What kind of a repository would be most valuable in supporting the reuse process.
- We should explore how the different reuse-oriented activities (e.g. managing the reuse infrastructure, producing, brokering, and consuming reusable assets) can be adapted to a software development process.

These three themes are the most important ones seen in the light of our research. Because we based our analysis on a known model of a software process (Lim's model), some tasks do not obtained attention very much although we see them as important parts of software process. This concerns especially testing and adapting of components. The current methods offer no component-specific means to test a software product or a piece of it. This issue deserves more efforts by the researchers.

Acknowledgements

This research was supported by Tekes[1] and companies participating in the PISKO-project. We also like to thank the four anonymous reviewers for their useful comments.

[1] National Technology Agency, Finland. `http://www.tekes.fi`

Bibliography

[1] Bailey, J., Basili, V.: The software-cycle model for re-engineering and reuse. Proceedings of the conference on Ada: today's accomplishments; tomorrow's expectations, ACM (1991), 267-281.

[2] Basili, V.: Facts and myths affecting software reuse. Proceedings of the 16th International Conference on Software Engineering (1994), 269.

[3] Basili, V., Caldiera G., Cantone G.: A reference architecture for the component factory. ACM Transactions on Software Engineering and Methodology, Vol. 1, No. 1, January (1992), 53-80.

[4] Batory, D., O'Malley, S.: The Design and Implementation of hierarchical software systems with reusable components. ACM Transactions on Software Engineering and Methodology, Vol. 1, No. 4, October (1992), 355-398.

[5] Biggerstaff, T., Richter, C.: Reusability framework, assessment, and directions. IEEE Software, March (1987), 41-49.

[6] Burton, B., Aragon, R., Bailey, S., Koehler, K., Mayes, L.: The reusable software library. IEEE Software, July (1987), 25-33.

[7] Caldiera, G., Basili, V.: Identifying and qualifying reusable software components. IEEE Computer, February (1991), 61-70.

[8] Capretz, L.: A CASE of reusability. Journal of Object-Oriented Programming, June (1998), 32-37.

[9] Davis, J., Morgan, T.: Object-oriented development at Brooklyn Union Gas. IEEE Software, January (1993), 67-74.

[10] Devanbu, P., Brachman, R., Selfridge, P., Ballard, B.: LaSSIE: A knowledge -based software information system. Communications of ACM, Vol. 34, No. 5, May (1991), 33-49.

[11] D'Souza, D., Wills, A.: Objects, Components, and Frameworks with UML: The Catalysis Approach. Addison-Wesley (1999).

[12] Dusink, L, van Katwijk, J.: Reuse dimensions. Software Engineering Notes, August (1995), Proceedings of the Symposium on Software Reusability, Seattle, Washington, April 28-30 (1995), 137-149.

[13] Fisher, G.: Cognitive view of reuse and redesign. IEEE Software, July (1987), 61-72.

[14] Henninger, S.: An evolutionary approach to constructing effective software reuse repositories. ACM Transactions on Software Engineering and Methodology, Vol. 6, No. 2, April (1997), 111-140.

[15] Horowitz, E., Munson, J.: An expansive view of reusable software. In Biggerstaff, T., Perlis, A. (eds.): Software Reusability, Volume I: Concepts and Models. ACM Press (1989), 19-41.

[16] Jaaksi, A., Aalto, J-M., Aalto, A., Vättö, K.: Tried & True Object Development: Industry-Proven Approaches with UML. Cambridge University Press (1999).

[17] Jacobson, I., Griss, M., Jonsson, P.: Software Reuse: Architecture, Process and Organization for Business Success. ACM Press (1997).

[18] Jacobson, I., Booch, G., Rumbaugh, J.: The Unified Software Development Process. Addison-Wesley (1999).

[19] Karlson, E.: Software Reuse: A Holistic Approach. John Wiley & Sons Ltd., (1995).

[20] Krueger, C.: Software reuse. ACM Computing Surveys, Vol. 24, No. 2, June (1992), 131-183.

[21] Lanergan, R., Grasso, C.: Software engineering with reusable designs and code. In Biggerstaff, T., Perlis, A. (eds.): Software Reusability, Volume II: Applications and Experience. ACM Press (1989), 187-195.

[22] Lenz, M., Schmid H., Wolf, P.: Software reuse through building blocks. IEEE Software, July (1987) 35-42.

[23] Lim, W.: Managing Software Reuse. Prentice Hall PTR (1998).

[24] Neighbors, J.: Draco: A method for engineering reusable software systems. In Biggerstaff, T., Perlis, A. (eds.): Software Reusability, Volume I: Concepts and Models. ACM Press (1989), 295-319.

[25] Nierstrasz, O., Meijler, D.: Research directions in software composition. ACM Computing Surveys, Vol. 27, No. 2, June (1995), 262-264.

[26] Ostertag, E., Prieto-Díaz, R., Braun C.: Computing similarity in a reuse library system: An AI-based approach. ACM Transactions on Software Engineering and Methodology, Vol. 1, No. 3, July (1992), 205-228.

[27] Prieto-Díaz, R.: Implementing faceted classification for software reuse. Communications of the ACM, Vol. 34, No. 5, May (1991), 88-97.

[28] Prieto-Díaz, R., Freeman, P.: Classifying software for reusability. IEEE Software, January (1987), 6-16.

[29] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W.: Object-Oriented Modeling and Design. Prentice-Hall, Inc., (1991).

[30] Sommerville, I.: Software Engineering Fourth Edition. Addison-Wesley, (1992).

[31] STARS Conceptual Framework for Reuse Processes (CFPR) Volume I: Definition, Version 3.0. STARS-VC-A018/001/00, October 25 (1993). Available at http://direct.asset.com/wsrd/product.asp?pf_id=ASSET%5FA%5F495.

[32] Taivalsaari, A.: A Critical View of Inheritance and Reusability in Object-oriented Programming. Ph.D. Dissertation, Jyväskylä Studies in Computer Science, Economics and Statistics, No. 23, University of Jyväskylä (1993).

Appendix 1. The More Detailed Analysis of Catalysis, OMT++ and Unified Process.

TABLE 3.4 More detailed results of the analysis of Catalysis.

Activity / Tasks	Catalysis
Producing Reusable Assets (PRA)	
Analyzing domain	Domain analysis is performed by creating business models. Business models include use cases, joint actions and type models. In this phase vocabulary of the domain is created with the aid of the user.
Producing Assets • Prefabricating • Retrofitting	Components are mainly created from the domain model and are designed for reuse. There is briefly mentioned reengineering existing systems. If third party components or legacy systems are used, one should create type models of them. Brief description of component development. Description of components is done by type models which are created during business modeling. Basic course of action is as follows: 1. Describe every component from the one users point of view. 2. Combine view points. 3. Design largest components and according to the use cases distribute them to functional blocks. 4. Define interfaces and extract business logic from user interface. Create layered architecture for the software. 5. Extract middleware from business components. 6. Create draft design model where one class presents one type. 7. Distribute responsibilities and collaboration among objects so that you can create flexible design. 8. Define components connections, and attributes and document them.
Asset • Maintenance • Enhancement	-
Brokering Reusable Assets (BRA)	
Assessing Assets	-
Procuring Assets	-
Certifying Assets	-
Adding Assets	Component management should be created through reuse group. Resources to build and maintenance repository should be given.
Deleting Assets	-
Consuming Reusable Assets (CRA)	
	Continues on next page

Continued from previous page (Catalysis)

Activity / Tasks	Explanation
Identifying • System • Assets	Software system is identified through use cases (joint actions). First one should define technical architecture which includes infrastructure components and their relationships with physical and logical architectures.
Locating Assets	-
Assessing Assets for Consumption	First architecture is implemented which guides experienced developer to find components.
Adapting / Modifying Assets	Components interfaces can be altered or interfaces can be added, but the component should be left unchanged.
Integrating / Incorporating Assets	-

TABLE 3.5 More detailed results of the analysis of OMT++.

Activity / Tasks	OMT++
Producing Reusable Assets (PRA)	
Analyzing domain	You model domain by class diagram and that gives you and your customer common vocabulary. Use cases and analysis phase's class diagram are also used to gather understanding of the context.
Producing Assets • Prefabricating • Retrofitting	Components which are created through OMT++ are prefabricated components. No discussion of retrofitting is done. Components are identified during architectural design which is done via OMT++ own 3+1 views. Components which result from this process are code components. Component identification process is basically as follows: 1. Create software architecture according to OMT++'s 3+1 views to the software architecture. As an result you get the following layers to the software system: System Products, Application products / Platforms; Applications / Service Blocks; Components; Classes 2. Implement group of classes as components. Component is 'size of the human being' (500 to 15 000 lines of code) and is a code component.
Asset • Maintenance • Enhancement	Component maintenance is when you create better (in size, functionality, quality etc.) component for the new software system and replace old component with it. OMT++ sees this as normal development work.
Brokering Reusable Assets (BRA)	
Assessing Assets	-
Procuring Assets	-
Certifying Assets	-
Adding Assets	-
Deleting Assets	-
Consuming Reusable Assets (CRA)	
Identifying • System • Assets	Software system is identified through use cases and you create the software architecture according to the use cases. Reusable components are discovered from the old system by experienced developers.
Locating Assets	Experienced developer locates reusable components from existing software system. He/she can use architecture descriptions to identify reusable components.
Assessing Assets for Consumption	-
Adapting / Modifying Assets	Guided by the software architecture (implicitly).
Integrating / Incorporating Assets	-

TABLE 3.6 More detailed results of the analysis of Unified Process.

Activity / Tasks	Unified Process
Producing Reusable Assets (PRA)	
Analyzing domain	RUP advises to do domain analysis as a project of its own. DA can be done via business modeling or domain modeling (which is subset of business modeling).
Producing Assets • Prefabricating • Retrofitting	RUP concentrates to prefabricate components and typical components are binary or executable files, files, libraries, table, or document. Application-specific and application-general layers are separated from middleware and system-software layers at the analysis phase. In analysis main focus is describing application layers and in design the focus is describing more middleware and system-software. Basic course of action to create component: 1. Create use cases. 2. Do analysis phase's class diagram and classify analysis classes to service packages. 3. Create service subsystems according to service packages. 4. Implement service subsystems. There should be straight mapping between service packages, service subsystems and implemented components. Components are identified and created by "reuse-enabled" developers.
Asset • Maintenance • Enhancement	-
Brokering Reusable Assets (BRA)	
Assessing Assets	-
Procuring Assets hline Certifying Assets	- -
Adding Assets	-
Deleting Assets	-
Consuming Reusable Assets (CRA)	
Identifying • System • Assets	Software system is identified through business/domain models and use cases. Mock up user interface can be used to gather more specific information about requirements. Existing systems can be used as a basis for identifying assets.
Locating Assets	Components are located through existing systems or by "reuse-enabled" developers. Possible components can be found from third party, through corporate standards for using and creating components (i.e. frameworks or design patterns), or by using "team memory".
Assessing Assets for Consumption	-
Adapting / Modifying Assets	Guided by software architecture (implicitly).
	Continues on next page

Activity / Tasks	Explanation *Continued from previous page (Unified Process)*
Integrating / Incorporating Assets	-

4 A MODEST BUT PRACTICAL SOFTWARE PROCESS MODELING TECHNIQUE FOR SOFTWARE PROCESS IMPROVEMENT

Ahonen, J.[†], Forsell, M.[‡], Taskinen, S-K.[†], "A Modest but Practical Software Process Modeling Technique for Software Process Improvement". This paper has been submitted for publication to Software Process: Improvement and Practice. Copyright may be transferred without further notice and the accepted version may be posted by the publisher.

[†] Information Technology Research Institute, University of Jyväskylä, Finland.
[‡] Chydenius Institute, University of Jyväskylä, Finland.

Abstract

One of the main problems with software engineering is due to the difficulties in evaluating and improving our software processes, especially in the light of the fact that reuse depends on a process which supports it. Generally used approaches to the evaluation and improvement of software processes are based on CMM, for example. In this paper we present a technique to improve software processes through modeling and evaluation. The presented technique is fairly easy to use, provides reasonably good results and requires only a fraction of resources required by CMM appraisals.

Keywords: process modeling, software process improvement, modeling technique

4.1 Introduction

Software companies are constantly fighting against the perils of software projects. Timetables do not hold, talented personnel is not available when required, budgets are overrun and software quality is not adequate. Those problems are severe and it is obvious that there are no "silver bullets" for those problems (Brooks, 1987).

The problems that software engineering organizations face are not, however, completely incurable. It is a common opinion that it is possible to improve the situation through gradual improvement (see e.g. Humphrey, 1989). One approach is to improve the software development process. The main ways to improve a process are

1. to omit unnecessary phases from the process;
2. to introduce new phases to the process; and
3. to improve the existing phases of the process.

The necessary steps to be taken in order to implement any of the above ways of process improvement require good knowledge of the actual software process to be improved. In this paper the terms *software*, and *software process*, and *software process modeling* are considered in the light of (Pressman, 2000, Boehm, 1976, and Gibson, 1999, and Curtis *et al.* 1992), respectively.

The software process is described (or prescribed) in extensive models like CMM (Paulk *et al.* 1993, Paulk *et al.* 1995), SPICE (SPICE, 2001) and ISO9001 (Kehoe & Jarvis, 1996). Those models do not include any way to model the actual process and for that purpose it is possible to use techniques like SADT, IDEF0 and Petri Nets (see e.g. (Curtis *et al.* 1992) for an evaluation). There are very few techniques which include processes, modeling and documentation in a single package (Curtis *et al.* 1992). In this paper such a technique is presented.

The technique has been developed in an industry cooperation project called PISKO. The aim of the project was to improve the software process — and especially software reuse — of the participating companies. The interest was especially in software reuse and the organization of reuse, in which the actual characteristics of the software process is especially important. This is expressed by Lim (Lim, 1997) when he stresses the importance of considering reuse in every phase of the software process, i.e. the actual characteristics of the process.

In the very first steps of the research it was decided that CMM is not probably useful considering the aim of the project and the lack of explicit notion of software reuse in CMM. The view was backed up by the companies which were already familiar with CMM. One of the problems expressed by the companies already familiar with CMM were the overall feeling of CMM appraisals. The representatives of those companies thought those appraisals too serious, almost like inspections in the army[1], and an approach with a more positive feeling was sought after.

[1] Something like: "All you programmers, arrange yourself in straight lines according to the quality manual. Now we will have a look at your code and your ways to program it. No speaking in the line."

In order to improve software reuse in participating companies a technique to model software processes was required. There were a few requirements for the technique. Those requirements were based on the fact that all participating companies were in constant need of talented personnel and their current employees were generally overstressed. Therefore any technique had to fulfill the following restrictions:

1. the technique should be easy to use for people who have no prior knowledge of it;
2. the technique should require as minimal resources as possible from the target organization; and
3. the technique should be able to point out real problems in the software process and hence provide input for process improvement.

In this paper a technique which satisfies the requirements in a reasonable way is presented. The situation in which the technique has been applied is outlined and the results are analyzed and their usability discussed.

4.2 PISKO Process Modeling Technique

The practical requirements lined out above restricted the complexity of the techniques that we could use. Process modeling is based on observation and reporting (Verlage, 1997). Therefore the most important part of any attempt to model the actual software process should emphasize the opinion and experience of the experts of the target organization. In our case the object of description and analysis is the software process actually used in the organization — not a process found from quality handbooks.

Our organization have earlier used the wall-chart technique (Saaren-Seppälä, 1997)[2] and experiences have been positive. The wall-chart technique is an excellent technique when a group of specialists are required to communicate and achieve a consensus regarding the current situation in a tight timetable (Karjalainen *et al.* 2000). It is the backbone of our modeling technique.

The PISKO Modeling Technique includes six phases, which are shown in the figure 4.1. We acquire the current process from expert knowledge by using the wall-chart technique in sessions which are organized in the target organization. During these sessions problems and points of improvement are recorded. After this the process models are transformed into a digital form which is inspected by the target organization. Next the process and problems are analyzed and evaluated, and naturally these results are inspected. In the following sections the phases are briefly discussed.

[2] The wall-chart technique has been fairly popular in Finland since 1970's, especially in information system design.

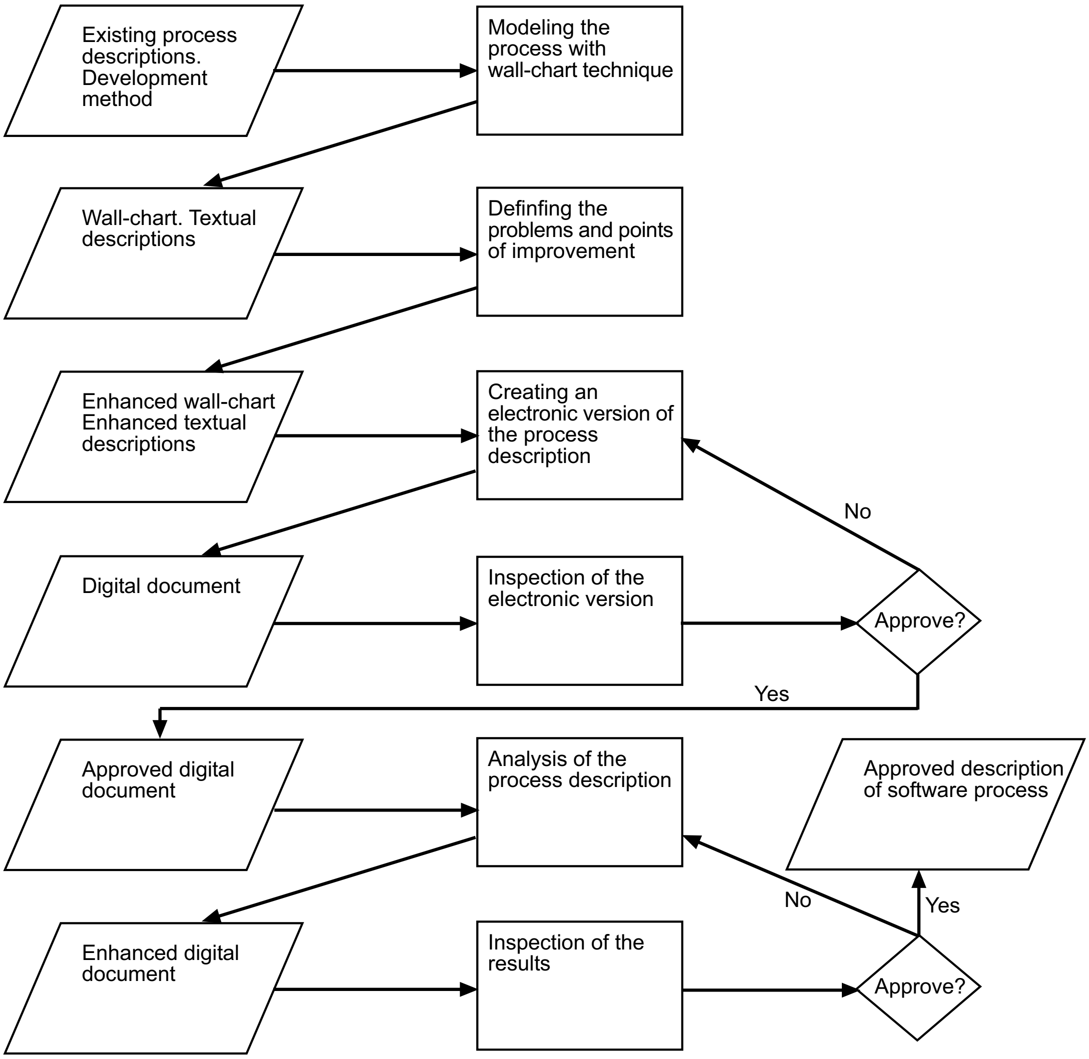

Existing process descriptions. Development method
Modeling the process with wall-chart technique
Wall-chart. Textual descriptions
Definfing the problems and points of improvement
Enhanced wall-chart Enhanced textual descriptions
Creating an electronic version of the process description
No
Digital document
Inspection of the electronic version
Approve?
Yes
Approved digital document
Analysis of the process description
Approved description of software process
No
Yes
Enhanced digital document
Inspection of the results
Approve?

FIGURE 4.1 The process description of the PISKO technique.

4.2.1 The Wall-Chart Sessions

During the Wall-Chart Sessions the model of the current process is constructed and problems with it are identified. The session lasts normally from three to five hours and it is done in a room where the session can be held without disturbance. In most cases there are seven people participating in the session: five experts from the target organization and two consultants. The experts should be chosen from those who know the actual software engineering process very well and have relatively long experience from the process. Two consultants are required to fill the roles of the chairman of the sessions and the secretary, whose responsibility is to make sure that every important piece of information revealed in the session will be written down.

The chairman and the secretary have prepared for the session by studying the existing material of the target organization's processes. That knowledge should, however, be used for gentle guidance, not for auditing. That is because the aim of the session is to let the experts describe the real process *as is*, which very often differs from the official process.[3]

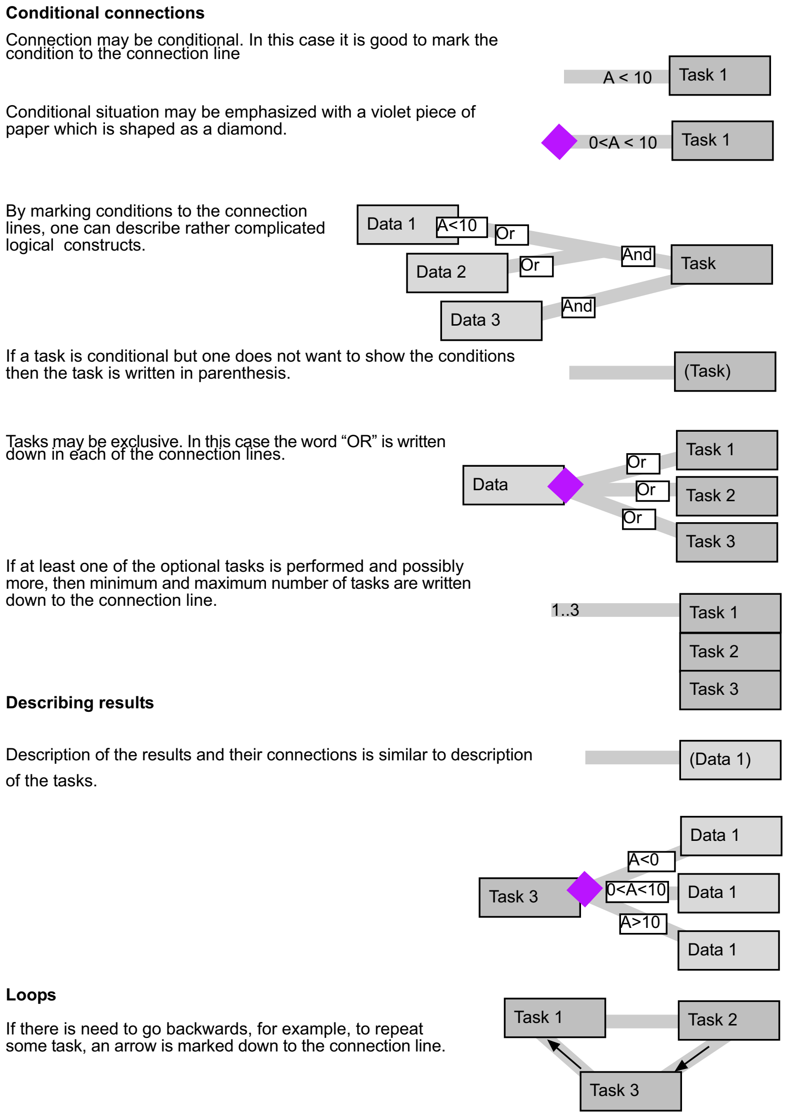
In the beginning of the session the technique is briefly explained to the experts. After the introduction the chairman lets the experts identify the main phases of the software process. Every expert is allowed, practically required, to express his/her opinion. When the experts have achieved a common opinion of a phase and its name, the secretary writes it down on a piece of paper. The chairman attaches the paper to the wall-chart into a place which represents the position of the phase in the software process. The flows of information are identified and represented in the same way (e.g. a document produced by a phase and required by another phase is represented by an elongated piece of paper which is marked by the name of the document). The wall-chart notation is shown in the figure 4.2, a simple process description is shown in the figure 4.3, and a wall-chart under construction is shown in the figure 4.4.

Often the process model needs greater accuracy. These may be pointed out by the experts participating the session, or the chairman may hint on the possible phases or flows that need to be handled with greater precision. Normally the experts can be trusted to make such decisions. In these cases the phases and flows of information may be divided into subphases and subflows if necessary. In some cases the subphases and subflows may require separate wall-charts and they need to be modeled in separate sessions.

The session may be recorded, but in many cases recording makes the situation a bit awkward. Therefore we have decided to have the secretary to make accurate notes of the discussion and especially on the reasons why particular decisions have been made and consensus achieved.

The secretary uses a predefined report template as much as possible during the note-taking. The outline of the template is shown in the figure 4.5.

[3] This is somewhat in contrast to the attitude of CMM appraisals.



Conditional connections

Connection may be conditional. In this case it is good to mark the condition to the connection line

Conditional situation may be emphasized with a violet piece of paper which is shaped as a diamond.

By marking conditions to the connection lines, one can describe rather complicated logical constructs.

If a task is conditional but one does not want to show the conditions then the task is written in parenthesis.

Tasks may be exclusive. In this case the word "OR" is written down in each of the connection lines.

If at least one of the optional tasks is performed and possibly more, then minimum and maximum number of tasks are written down to the connection line.

Describing results

Description of the results and their connections is similar to description of the tasks.

Loops

If there is need to go backwards, for example, to repeat some task, an arrow is marked down to the connection line.

FIGURE 4.2 The wall-chart notation.

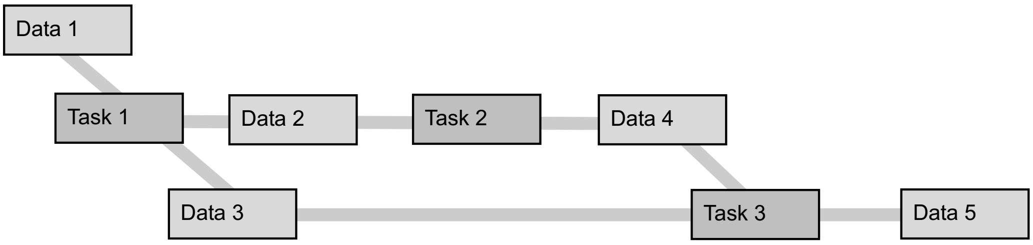


FIGURE 4.3 A process fragment.

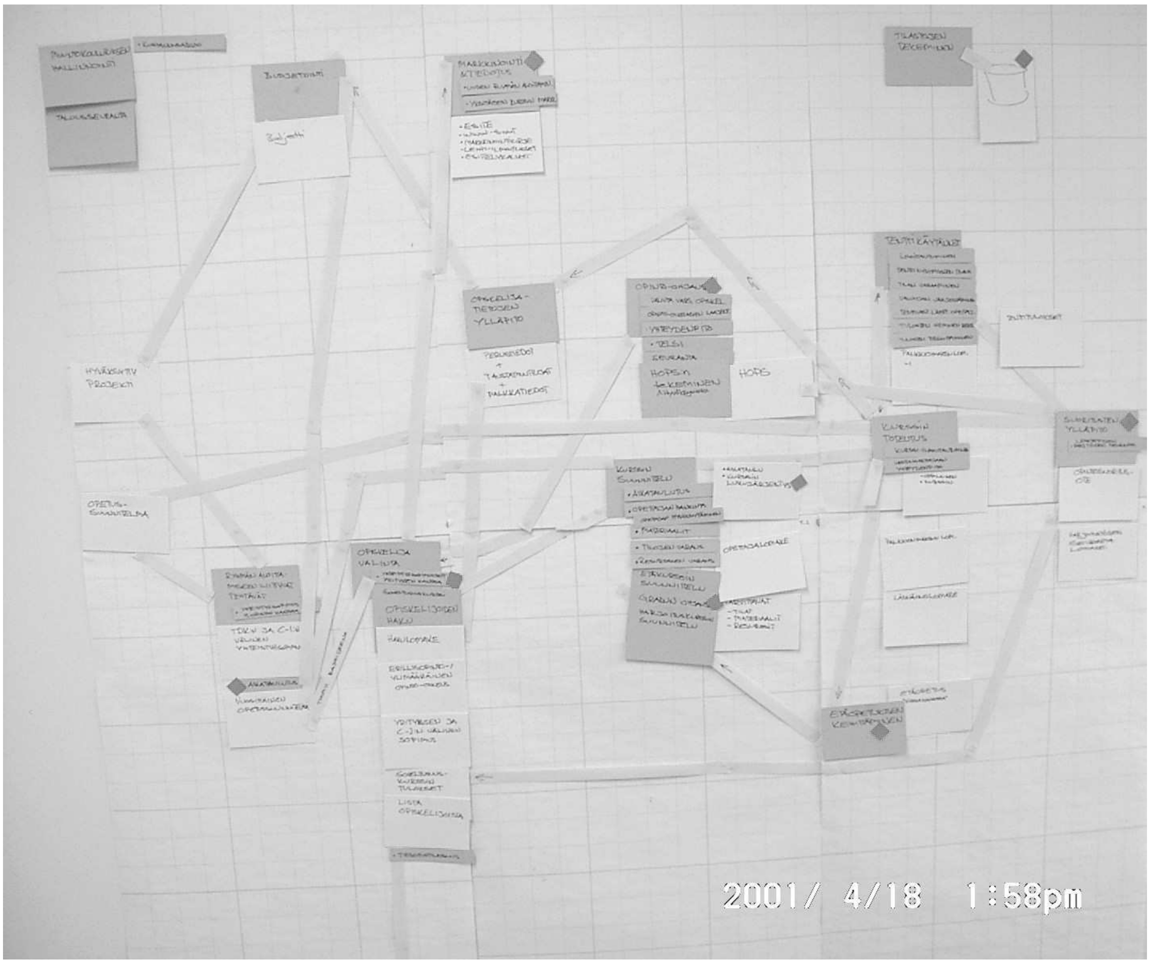


FIGURE 4.4 A wall-chart under construction.

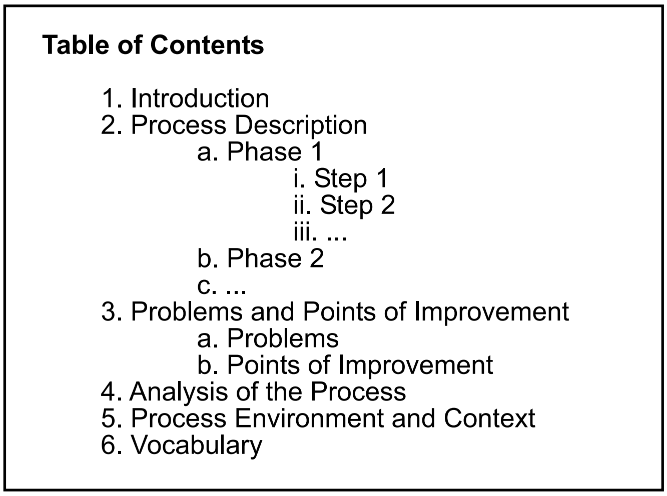


Table of Contents

1. Introduction
2. Process Description
    a. Phase 1
        i. Step 1
        ii. Step 2
        iii. ...
    b. Phase 2
    c. ...
3. Problems and Points of Improvement
    a. Problems
    b. Points of Improvement
4. Analysis of the Process
5. Process Environment and Context
6. Vocabulary

FIGURE 4.5 The table of contents for the process description document.

The result of the first phase is a wall-chart presenting the software process *as is*. Also, the secretary have taken notes about the comments of the experts, that they have made during the modeling session.

4.2.2 Problem Definitions

The second phase of the modeling session finds and outlines the problems existing in the software process. Most of the problems have been found out during the modeling phase when the experts try to find a consensus regarding the process, specific phases, their control, output and/or input. In addition to those spontaneously found problems, the chairman and the secretary use their prior knowledge and the notes in order to assure that all relevant problems are brought up, identified and marked by using the wall-chart notation.

Each problem is named and numbered when identified. The number is written down to a red piece of paper which denotes the problem in the wall-chart. The experts locate the phase or information flow where the problem occurs. The experts describe the problem and its significance, and they identify phases the problem affects. This phase results list of problems that are numbered, described and attached to the wall-chart.

4.2.3 Process Documentation and Inspection of Documentation

The wall-chart session ends when the experts think that the achieved accuracy is good enough. After the decision the experts return to their normal work, but the secretary and the chairman start documenting of the session. In order to create the documentation the wall-chart is photographed (the figure 4.4 is one of the photographs taken). There should be enough photographs so that detailed analysis of the wall-chart is possible even without the physical chart. The physical chart is carefully packaged and the consultants return to their office.

The consultants create a digital version of the wall-chart and of the notes of the secretary. They aim to a coherent and understandable representation. The process description is refined and it is completed with textual descriptions. The process description notation is different than in wall-chart due to the requirements of understandability and expressivity.

The digital process description is sent to the target organization for inspection. Misunderstandings, errors and omissions are clarified and corrected according to the feedback provided by the target organization's experts. The resulting document is an approved description of the organization's software process.

4.2.4 Analyzing the Process and Inspecting Results

The approved digital model of the process is used for analyzing and evaluating the process. The chairman and the secretary carefully analyze the process. In the analysis, specific consideration is paid to the problems in the software process.

The analysis should consider existing process models and standards like CMM (Paulk *et al.* 1993, Paulk *et al.* 1995), SPICE (SPICE, 2001) and ISO9000 (Kehoe & Jarvis, 1996). Common sense should be used in evaluating and analyzing the process (see e.g. Paulk, 1999), and there is no reason why best ideas from CMM, SPICE and ISO9000 should not be utilized. The result from the analysis is the completed report (the layout of the report is outlined in the figure 4.5) of the process and a list of suggestions for improvement.

The completed report is evaluated and commented by the experts of the target organization. The report is corrected if necessary for approval. After approval the target organization can use the report and its suggestions in order to improve its software process.

4.3 Evaluation of the Technique

The modeling technique and its application presented in the previous section require just a few resources. Therefore the practical usability of the technique requires further evaluation.

4.3.1 Background for Evaluation

The technique has been used in four different target organizations and for each organization three separate modeling sessions were held. The total number of modeling cases is twelve. The target organizations were different in their size and operative sector in the information technology market.

The first organization is a small software development company which produces software mainly as a subcontractor for an international company which produces telecommunications technology and applications. The number of software engineers and other development personnel is below one hundred. In most cases the company uses the same software engineering method as its main client. The method has been successfully used in several large-scale projects by both the client and the company itself. The method requires demanding education in order to be utilized correctly. The company grows rapidly which creates new requirements for its processes constantly.

The second organization is a small software company with less than fifty software developers. The main product of the company is a family of mobile terminals for logistics and transportation. The product family includes a few different product lines with their own operating systems, programming APIs, development tools, and programming guidelines for their customers. The method used is a structural approach to embedded systems programming and design, although it has been gradually enhanced during the last few years. During the time of the study reported in this article the process was managed by senior engineers without a formal guidance. Because the company had started to grow at a reasonable rate, the need for formal process definition and general improvement arose.

The third company is a medium sized company producing automation hardware and software for the utilization and management of their automation systems. The company has less than one thousand software engineers. The process used by the company has its strengths in managing large-scale system deliveries. The driving force of that in-house developed method is that the results produced by subphases must be useful in subsequent phases. Therefore reuse, change management, and testing are especially important for that company.

The fourth company is a large software company which produces shrink-wrapped and tailored software for its customers. The company has approximately seven thousand developers. The company has its own in-house developed method which incorporates both object-oriented and structural development.

4.3.2 Evaluation

The evaluation of the modeling technique is based on the answers given to a questionnaire presented in appendix (the appendix includes a summary of the answers). One of the aims of the questionnaire was to separate modeling as a method and the used technique in order to be able to evaluate them both. It must be noted, however, that there were only six answers to the questionnaire, and that should be kept in mind during the evaluation.

Process modeling as a method got positive reactions. Modeling was seen as a way to improve the understanding of the process and the effort used for it was well-spent. Modeling was considered a suitable method to be used in identifying problems and potential points of improvement.

The wall-chart technique was seen as useful in creating process models which represent the real process. The technique permitted experts to describe and explain all important phases, information flows, and tasks in the process. In addition, the most dominant opinion was that the technique succeeded in making problems visible, but it did not place the problems or points of improvement in any useful order or classified them according to how remarkable they are. The technique did not identify those parts of the process which were strong.

Generally the modeling activity with wall-chart technique was considered too dependent on the chairman's and the secretary's professional knowledge and social capabilities. Further, the technique was considered very dependent on the actual expertize of the experts chosen to represent the target organization. One of the problems seen with the technique was that it did not produce practical proposals on the actual implementation of improvements or changes in the case that the chairman and the secretary were not resourceful enough.

It was not considered necessary for the experts to have prior knowledge of the wall-chart technique. The time required by the sessions or the number of experts participating in the session were not considered important. The optimum number of experts is probably five or six because the effectiveness of wall-chart technique slows down with larger numbers (Saaren-Seppälä, 1997), but it has not been tested because no more than six experts participated in any session.

According to the results the technique can be used to:

- increase the knowledge of the real process;
- model the current software process;
- identify the points of improvement; and
- propose improvement points.

The results can be achieved with a reasonably small investment in required resources.

4.4 Discussion

According to the answers and other experience from the technique, it is suitable for modeling the software process and identifying points of improvement and possible problems. It does not, however, give any order in which improvements should be conducted. The target organizations have implemented some improvements, but not in the expected scale. Reasons for that include the lack of any particular priority of the improvements and the lack of resources.

It is interesting to compare the results to the results of the study conducted by Goldenson and Herbsleb (Goldenson&Hersleb, 1995) in which they studied how CMM-based appraisals are perceived by different organizations and how those appraisals have changed the organization. Their study is much broader, but it is very interesting to compare the results.

According to Goldenson and Herbsleb (Goldenson&Hersleb, 1995) the problems of the process identified with CMM-based appraisals were considered generally correct but the appraisals were unable to order the importance of the problems. Role of the conductors of the CMM-based appraisal was considered important although some reported that their role was negative. However, organizations think that the CMM-based appraisals give usable results and they are used as bases to initiate process improvements. Problems are that CMM-based appraisals do not give sufficient information to conduct the improvements and they do not support improvements. Respondents considered CMM appraisals more like road maps and conducting the actual implementation of improvements require more specific guidance. This is supported by Paulk (Paulk, 1999) when he states that CMM is road map and it does not prescribe the change. In addition to that, some of the improvements are clearly too ambitious and initiating them becomes harder (Paulk, 1999).

When comparing the results of CMM-based appraisals and the results from the PISKO Modeling Technique interestingly enough the results are roughly comparable. This is especially interesting due to the fact that the PISKO Modeling Technique requires very little resources from the target organization when compared to the resources required by CMM.

Generally it seems that the use of resources required by CMM are an overkill. Fairly similar results can be achieved by the PISKO Process Modeling Technique which is easy to use, does not require extensive use of resources from the target organization and has been shown to be useful in software process improvement. The missing features are the ability to order possible improvement points and how to implement those improvements. In addition to that, the notion of different perspectives to the process should be included to the technique, as proposed in (Curtis *et al.* 1992). The technique should be improved to include the missing features.

References

Boehm, B. 1976. Software Engineering. *IEEE Transactions on Computers* C-25(12): 1226-1241.

Brooks, F. 1987. No Silver Bullet - Essence and Accidents of Software Engineering. *Computer*, April: 10-19.

Curtis, B., Kellner, M., Over, J. 1992. Process Modeling. *Communications of the ACM* 35(9): 75-90.

Gibson, R. 1999. Software Process Modeling. In McGuire, E. (ed.) *Software Process Improvement: Concepts and Practices.* Idea Group Publishing: Hershey, PA; 1-16.

Goldenson, D., Herbsleb, J. 1995. *After the Appraisal: A Systematic Survey of Process Improvement, its Benefits, and Factors that Influence Success.* Technical Report CMU/SEI-95-TR-009.

Humphrey, W. 1989. *Managing the Software Process.* Addison-Wesley.

Kehoe, R., Jarvis, A. 1996. *ISO 9000-3: A Tool for Software Product and Process Improvement.* Springer-Verlag: New York.

Lim, W. 1997. *Management of Software Reuse.* Addison-Wesley: Upper Saddle River, NJ.

Paulk, M., Curtis, W., Chrissis, M., Weber, C. 1993. *Capability Maturity Model for Software, Version 1.1. Technical Report.* CMU/SEI-93-TR-24, DTIC ADA263404.

Paulk, M., Weber, C., Curtis, B., Chrissis, M. 1995. *The Capability Maturity Model: Guidelines for Improving the Software Process.* Addison-Wesley: Reading Massachusetts.

Paulk, M. 1999. Using the Software CMM with Small Projects and Small Organizations. In McGuire, E. (ed.) *Softawre Process Improvement: Concepts and Practices.* Idea Group Publishing: Hershey, PA; 76-92.

Karjalainen, A., Päivärinta, T., Tyrväinen, P., Rajala, J. 2000. Genre-Based Metadata for Enterprise Document Management. *Proceedings of the 33rd Annual Hawaii Interna-tional Conference on System Sciences* (HICSS); Digital Documents Track; Genre in Digital Documents, Maui HI, USA, January 4-7, IEEE Computer Society, Los Alamitos CA, CD-ROM.

Pressman, R. 2000. *Software Engineering: A Practitioner's Approach, Europiean Adaptation, Fifth Edition.* McGraw-Hill International (UK) Limited.

Saaren-Seppälä, K. 1997. *Seinätekniikka prosessien kehittämisessä.* (Using the wall-chart technique in process development, in Finnish), Kari Saaren-Seppälä Ltd., Finland.

SPICE. 2001. http://www.sqi.gu.edu.au/spice/, visited 31.8.2001.

Taskinen, S-K. 2000. *Prosessin mallinnusmenetelmän soveltuvuus ohjelmistotuotantopro-sessin kehittämisen apuvälineeksi* (Applicability of software process modeling to support software process improvement, in Finnish). Master's Thesis, University of Jyväskylä.

Verlage, M., Experience with Software Process Modeling. *Software Process: Improvement and Practice* 3(2): 133-136.

APPENDIX: QUESTIONNAIRE

A BACKGROUND

1. Resources needed to modeling session
   1.1 Name of the company:
   1.2 Time used to modeling and inspections:
   1.3 What were the main goals of the modeling:
   1.4 Did the goals change during modeling session and how?

B QUESTIONS ABOUT THE TECHNIQUE

2. General valuation: PISKO Process Modeling Technique in general. Do you agree or disagree with the statements?

	Question	Totally disagree	Disagree	Don't know	Agree	Totally agree
2.1.	Technique helped to improve the understanding of the current process.	0	2	1	3	0
2.2.	The results gained with the process modeling were well worth the effort we spent.	0	0	0	4	2
2.3.	Technique succeeds in identifying problems and points of improvement.	0	0	0	3	3
2.4.	Created process descriptions and documentation are adequate to perform the improvements.	0	3	0	3	0
2.5.	Technique helped to create a realistic process description.	0	0	0	4	2
2.6.	The created process descriptions were too dependent on the expertise and judgment of the experts attending to the modeling session.	0	2	2	2	0
2.7.	The reality and content of the process description is too dependent on the session leader's expertise and faculty to judge.	0	5	0	1	0
2.8.	Used modeling technique is not able to describe all the important aspects (phases and tasks inside those phases) of the process.	1	4	0	1	0
2.9.	Technique helps to figure out the sequence of the needed actions of improvement.	0	2	1	3	0
2.10.	Identified problems or points of improvement were essential.	0	0	0	4	2
2.11.	Suggested actions of improvement were good and it was/would have been possible to implement them.	0	0	1	4	1
2.12.	The used notation is able to represent all of the most important aspects of the software process.	0	0	0	5	1

3. Success factors: Attributes that are perceived to support utilizing the method.

	Question	Substantially	Moderate	Don't know	Some	Little if any
3.1.	The expertise and the competence of the leader of the session.	1	5	0	0	0
3.2.	Knowledge about the process the participants already have and the quality of that knowledge.	1	2	0	3	0
3.3.	Previously made process descriptions, documentation and definitions.	0	2	0	4	0
3.4.	The amount of information gathered during process modeling session.	2	3	0	1	0
3.5.	Objectives and their clarity.	2	3	0	1	0
3.6.	Approval of the results right after collecting them.	2	3	0	1	0
3.7.	The structure of the group that is using the technique.	2	3	1	0	0
3.8.	Competence of the participants.	2	3	1	0	0
3.9.	Time consumed to process modeling.	0	1	1	4	0
3.10.	Inspections after modeling session.	0	3	0	3	0
3.11.	The number of participants in the process modeling session.	0	0	3	3	0

C INFLUENCE OF THE PROCESS MODELING EFFORT:

4. Experiences after the modeling

Mark as many boxes as you need to describe you experiences and answer to the additional questions when needed.

□ After the process modeling the software process has changed. How?

"We recognized a part of the process that doesn't have an operations model. We have started to improve that part of the process."

"Small changes are happening every day because customers are changing and developing their own methods. The major change is the new phase in the process, architecture design, that includes component designing."

□ After the process modeling there have been changes mostly in the way of thinking. What kind of changes have happened?

"We have started to improve also the processes in the upper level, like the product management -process."

"We are going to add the points of reuse to our current process model."

□ Nothing much has changed since the process modeling. Possible reasons:

"Lack of time, no allocated resources to process improvement."

□ Results of the process modeling didn't answer to the objectives. Possible reasons for this:

- No marks here

□ After the modeling sessions we have seriously considered starting systematic process improvement actions. What kind of actions has been done to make the start of the improvement easier?

- Two of the companies have started to consider the process improvement but no actions were listed here.

- □ After the modeling session we have started our process improvement actions. What kind of actions has been done?

  "We have updated our quality manual."

- □ Other things took priority and we didn't start process improvement. What kind of events or crises there was?

  - No marks here

- □ After the modeling session we have been interested in process improvement, but to start the concrete actions is difficult.

  - One mark here, but no comments.

- □ Process modeling was just one task in PISKO-project and we didn't have great expectations or objectives concerning it.

  - One mark here, but no comments.

5. About the results

   5.1 How accurately did the technique identify the major problems with software process? What kind of problems left unidentified?
   "Problems were identified very well."
   "One important problem was found."
   "Different projects use different kind of process and they have different kinds of problems that only people who are involved are aware of. "

   5.2 Did the technique identify "needless" problems? Examples?
   "No "needless" problems were identified."
   "We recognized some "needless" problems."

   5.3 How well did the technique characterize the strong points of the organization and process?
   "Mostly the weak points were identified."
   "The opinion and estimations of the consult (session leader) were in important role in this."
   "The technique did not support the identification of the strong points."
   "The strong points were identified well enough."

   5.4 Were the improvement suggestions good enough to solve the identified problems?
   "We found points of improvement, not means to fix the problems. "
   "The suggestions of improvement were mainly ideas. "
   "Some, but not much."
   "The suggestions of improvement were quite approximate."

   5.5 Did the process descriptions help to prioritize the importance of the found problems or the points of improvement?
   - Two *"yes"* replies.
   - Two *"no"* replies.
   "They support at least the decision-making."

   5.6 Does the documentation and the analysis of the process contain useful additional information?
   - Three *"yes"* replies.
   - One *"no"* reply.
   - One *"some"* reply.

   5.7 Were the points of improvement and the solutions to them easy to recognize?
   - Four *"yes"* replies.
   "Quite easily."

5.8 Were the improvement suggestions proven to be practical and useful?
"The improvement effort is still unfinished."
"They are solutions to problems."
"Not exactly."

5.9 Were the improvement suggestions practiced in action?
- Two *"yes"* replies.
- One *"no"* reply.
"They will be."
"I don't know."

5.10 Were the gathered information shared among organization?
"Yes, we started to model the product management process based on that."
"Yes, to people responsible for process improvement tasks."

5.11 How well the used modeling technique was able to represent the aspects of the software process? What was missing?
- Three *"well"* replies.
"Process was well described and the need for improvement was recognized."
"Process was well described, but the phases and results should be represented more precisely. "
"It should be possible to describe the process more precisely to the A3-paper."

5 USING HIERARCHIES TO ADAPT DOMAIN ANALYSIS TO SOFTWARE DEVELOPMENT

Forsell, M.[†], "Using Hierarchies to Adapt Domain Analysis to Software Development". Sein, M., Munkvold, B., Ørvik, T., Wojtkowski, W., Wojtkowski, W. G., Zupančič, J. (eds.), Contemporary Trends in Systems Development. Papers presented at ISD2000, the Ninth International Conference on Information Systems Development: Methods and Tools, Theory and Practice, August 14-16, 2000, Kristiansand, Norway. Kluwer Academic/Plenum Publishers, New York, 2001, 105-118.

©Kluwer Academic/Plenum Publishers, 2001. Reprinted with permission.

[†]Information Technology Research Institute, University of Jyväskylä, Finland.

Abstract

Domain analysis is used to achieve reusability in software development but the current component-based software development methods do not treat domain analysis deeply enough and domains are seen quite narrowly in domain analysis methods. We present hierarchical domain analysis that helps adapting domain analysis to the software development and it also aids using reusable information across domains. Hierarchical domain analysis has phases to reuse domain analysis results across different domains and it is to be used with the company's current software development method.

5.1 Introduction

Software development strives toward increasing the amount and quality of the software and at the same time decreasing the costs and development time. One approach to achieve these diverse goals is the systematic software reuse (Biggerstaff and Richter, 1987). In the reuse-oriented software development the key success factor is domain analysis (DA) (Arango, 1989; Lam and McDermid, 1997; Prieto-Díaz, 1994). DA is a process through which information used in software development is identified, captured, and organized with the purpose of making it reusable when creating new systems (Prieto-Díaz, 1990). While the traditional development methods (e.g., Jaaksi et al., 1999; Jacobson et al., 1999) focus on one application, DA focuses on classes of applications (Arango, 1994).

Neighbors (1980; 1989) was one of the first persons to introduce DA as part of software development. DA was used to gather necessary information to be used in the DRACO system, which created software by assembling existing components. Since then many DA methods have been introduced (see e.g., Prieto-Díaz and Arango, 1991; Wartik and Prieto-Díaz, 1992; Arango, 1994) and they have been used to produce domain specific application generators (Villarreal and Batory, 1997), domain specific application frameworks (Codenie et al., 1997), and domain specific software architectures (Tracz, 1995), among other things. In most examples domains are quite narrowly understood, for example, text formatting domain (Arango and Prieto-Díaz, 1991), airline reservation domain (Prieto-Díaz, 1987), or database domain (Villarreal and Batory, 1997). Hence, the reuse that DA fosters in these cases resides inside these narrow or small domains. Furthermore, DA is most often seen as viable inside domains that are mature (Wartik and Prieto-Díaz, 1992).

Current component-based development methods (Jaaksi et al., 1999; Jacobson et al., 1999; D´Souza and Wills, 1999) use DA as one part of the development work (Forsell et al., 2000). Unfortunately, DA is only briefly introduced and it is used to identify possible components inside one application. The components in these methods help manage the development work, and enable replacing components so that changes do not propagate to the rest of the software. Our problem here is twofold. On one hand, a development method needs to use components as available resources, not only as a means to control the development or maintenance work. On the other hand, DA needs to be part of the whole company's software development, not only inside narrow highly specified domains. Based on these observations we specify our research question as follows: ″How should the domain(s) be analyzed so that all solutions readily available could be part of the resulting software solution, and that results can be produced and used in everyday software development?″

We assume here that it is feasible to use DA to achieve answer to this question. And in this paper we illustrate how these problems can be tackled with the hierarchical domain analysis. We show that by conceiving all the software a company produces as a domain, and by dividing this ″big″ domain into sub-domains, we can use results from DA in software projects that traditionally would be considered as being in different domains (in our case they are in the different sub-domains). The approach in this paper is created for TietoEnator Corporation (the largest software company in the Scandinavia) and it is meant to be used as one part of TietoEnator's in-house software development method TietoObject.

The structure of the paper is as follows: In Section 2 we introduce key concepts of DA and present Common Process (Arango, 1994) to perform DA. In Section 3 we introduce hierarchical domain analysis concepts. In Section 4 we present hierarchical domain analysis process and show how it relates to Common Process. Also, phases of hierarchical domain analysis are described more closely. Finally, in Section 5 we discuss about the hierarchical domain analysis, present limitations for this study and set out some further research questions.

5.2 Background for Domain Analysis

5.2.1 Domain Analysis Concepts

The usefulness of DA is grounded on the belief that reusable information is dependent on the problem domain, and that the problem domain is cohesive and stable (Arango and Prieto-Díaz, 1991). Arango and Prieto-Díaz (1991) see a problem domain as a synonym for a class of problems that are similar. They add that in software context a body of information is considered a problem domain if:

1. deep or comprehensive relationships among the items of information are known or are suspected with respect to some class of problems,
2. there is a community that has a stake in solving the problems,
3. the community seeks software-intensive solutions to these problems, and
4. the community has access to knowledge that can be applied to solving the problems.

Besides the problem domain we also have a solution space. A solution space is the area inside of which a solution to the defined problems has to be created (Tracz, 1995). In a software-intensive solution we have to use, by definition, software at least partly in our solution. The solution space tells us what kind of operating systems, databases, object request brokers, and programming languages, among other things, we have to use when a solution to the defined problems is designed.

The result of DA is a domain model. In the domain model the concepts and the relations between these concepts are presented and the rationale for them is given (Arango and Prieto-Díaz, 1991). It is quite difficult to draw a line between domain modeling and domain analysis and in many cases they are used as synonyms (Arango, 1994). We use the term domain analysis, and we consider that it is hard to do analysis without modeling the results of it.

5.2.2 Common Domain Analysis Process

Arango (1994) argues that all domain analysis methods he evaluated map onto Common Process. We argue that this also holds true for Tracz's DSSA (Tracz,

1995) and the newest version of Organizational Domain Modeling by Simos (1996) although they were not among those evaluated. The five generic activities of Common Process are:

1. domain characterization and project planning,
2. data collection,
3. data analysis,
4. classification, and
5. evaluation of domain model

In the domain characterization and project planning activity the necessary preparation for DA is done. This means that it is decided if the DA is feasible in the first place. Also, the problem domain is identified and its boundary is set. One difficult part of DA is defining the boundary (Prieto-Díaz, 1987) where we have to make clear what is inside the particular domain and what is outside.

In the data collection activity all valid information about the domain is gathered. In order to do this information sources must be identified. Good sources are literature, domain experts, existing applications, customer surveys, and market analysis, among other things (Arango and Prieto-Díaz, 1991).

Gathered information is checked for correctness, consistency and completeness in the data analysis activity. Validated information is analyzed to find similarities and variation points between concepts, activities, and relationships. (Arango, 1994.) This kind of information modeling concerns classes of applications as the traditional analysis techniques in software development methods are used to analyze one application.

The next activity is classification and it is the most important phase of the DA. As a matter of fact, classification is the only thing that differentiates domain analysis from software development cycle analysis. Classification captures and makes explicit the information structures that characterize classes of applications in the domain. (Arango, 1994.)

The fifth generic activity is evaluation of the DA results. Here the produced models and results are inspected and checked for correctness and completeness. Often the most suitable evaluators for the results are the domain experts.

5.3 Hierarchical Domain Analysis Concepts

Hierarchy is one of the most often used concepts to give order for human thinking. Jarzabek (1997) outlined DA approach that identified multiple domains underlying a program family (cf. Parnas, 1976). This "divide and conquer approach," according to Jarzabek, makes results of DA more understandable and it facilitates design of a reference architecture for program families. We apply this idea of hierarchies into another direction, i.e., we see all the software that company produces as a domain and this company-wide domain can be further refined into sub-domain. So, we see domains as business areas and domains (or sub-domains) help achieve some business objectives for the company. This company-wide domain can be divided into sub-domains, which are subsets of the business area and together comprise the entire domain. The sub-domains may be comprised form other sub-domains. This way we can gradually refine business areas as they evolve during the lifetime of an organization. (See Wartik and Prieto-Diaz, 1992.) The aim of using hierarchies this way is to facilitate the reuse of software components in the whole company and not just inside a narrow domain. We call this approach hierarchical domain analysis (HDA).

The emphasis of DA has shifted from the reuse of the code to the reuse of more abstract structures (Wartik and Prieto-Díaz, 1992; cf. McIllroy, 1976). We do not want to hinder this progress and thus we see the components that can be found during DA as products of the software development process (see e.g., Freeman, 1983; Horowitz and Munson, 1984; Karlsson, 1995; Krueger, 1992; McCain, 1985; Wegner, 1983; Whittle, 1995). Software is not only the program code but also the documentation that supports the use of the software. This means that a software component may be requirement, design document, program code, test script, installation manual or a user's guide, among other things. However, we do not regard the software development process or methods used as software component.

In DA it is important to separate the problem domain from the solution space (see Jackson, 1994; Tracz, 1995). We see that the problem domain concerns the business of a company and the solution space deals with the constraints that are built by the software solution. In Figure 5.1 domain is partitioned to two aspects. On one hand, we have business aspect that deals with the problems in the domain. On the other hand, we have the solution space, which is comprised of the software that is used to solve the problems.

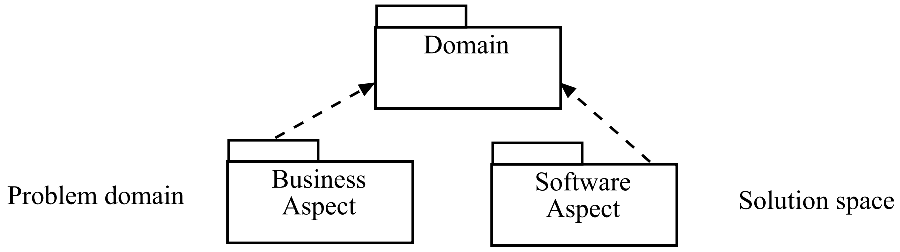


FIGURE 5.1 Basic Partitioning of the Hierarchical Domain Model into the Problem Domain and the Solution Space (UML Package Diagram).

We distinguish between four separate roles in the HDA process. The roles are a domain expert, an user of the software system, a domain analyzer and a (software) developer. A Domain expert is a person with extensive experience of, or substantial knowledge, about the work in the domain. A user is a person who actually works in the modeled domain and uses software that is created into that domain. A domain analyzer is someone who creates HDM and she or he must have knowledge about the technique. The domain analyzer is concerned with the similarities and the variation points in the domain. A developer uses the results from the HDA process, when she or he is creating new software (from components) to the domain. The domain analyzer and the developer might be the same person in small companies but in these cases one has to make it clear which role she or he plays (cf. Moore and Bailin, 1991).

5.4 Hierarchical Domain Analysis Process

We use Common Process as a starting point but we add some tasks to it. When Common Process divides the process to activities we divide the process to phases. Every phase in HDA contains several steps. In Figure 5.2 relations between activities in Common Process and phases in HDA can be seen. As one can notice from Figure 5.2 we add one phase prior to any Common Process activities. In the first phase of HDA we create a domain model to cover all the software that the company produces and also, we model the software aspect that set limits to solutions. Secondly, we add a phase after all Common Process' activities. In the fifth phase of HDA we use the results in the current software development project. Third modification is that we use activities two to five of Common Process in HDA phases three and four. Fourth notable point of HDA that is not visible in the picture is that we do classification in two stages. First we classify data according to Common Process after which we have a second classification stage. This stage locates identified components into the company-wide domain model.

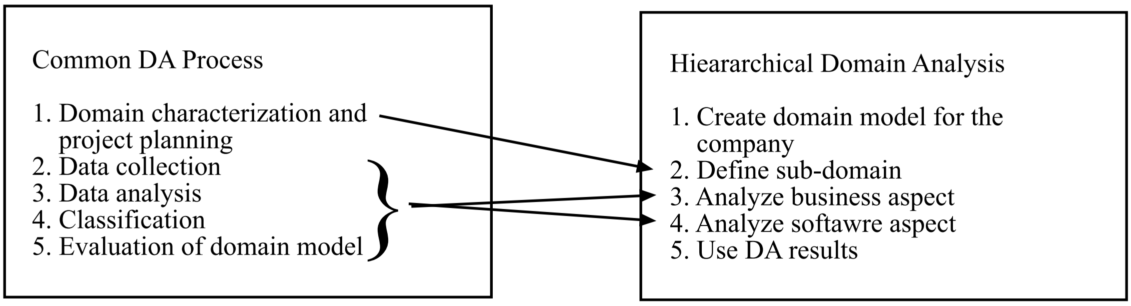

Common DA Process
1. Domain characterization and project planning
2. Data collection
3. Data analysis
4. Classification
5. Evaluation of domain model
Hieararchical Domain Analysis
1. Create domain model for the company
2. Define sub-domain
3. Analyze business aspect
4. Analyze softawre aspect
5. Use DA results

FIGURE 5.2 Relationships between Common DA Process and Hierarchical Domain Analysis.

HDA is used as part of an existing development method. Phases from one to four are done prior to the development method's analysis phase. Phase five is performed (at least partly) in all the development method's phases.

Next, we present the phases and steps of the HDA process. We present only briefly those phases and steps, which are well presented in Common Process and concentrate on the phases and tasks that differ from it. We chose to use Unified Modeling Language (UML) (Booch et. al., 1998) and its package diagram to present the results of the HDA, which is hierarchical domain model (HDM).

The reasons for selecting UML are due to its de facto status and our familiarity with it. We also use other UML diagrams during HDA such as the activity and class diagrams but they are in a subsidiary role (see Arango, 1994). Although we chose UML to be used as a modeling language we see that any modeling technique that offers similar features can be used to model HDM.

5.4.1 Create a Domain Model for the Company

The objective of the first phase of HDA is to create company-wide domain model, where the company is divided into hierarchical structure of sub-domains. The domain experts in this case are the top management of the company and the information system designers. One must notice that the first phase is performed in its total range only when HDA is used for the first time. When HDA is used in the subsequent software development projects, it is only checked for correctness, and only the parts that are particular to the development project at hand are scrutinized.

The first phase has four steps:

1. Identify company's key processes

2. Determine the rationale for software development

3. Create the hierarchy of sub-domains for the business aspect

4. Create the hierarchy of sub-domains for the software aspect

The first step identifies how the company accomplishes its business objectives. The answer to this can be found partly from the top management and partly from the organizational structure. We are assuming here that top management is aware of the company's mission and vision, and that they know what are the strategies to get there. Organizations group their units basically either by the function performed or the market served (Mintzberg, 1984) and this grouping is a fundamental means to coordinate work in the organization (Mintzberg, 1983). Software programs are produced for the company to achieve its business objectives. This means that a software program must give service that is useful for the organization.

The second step determines the rationale of the software development. The companies produce the software for various reasons. One can produce software for the in-house needs and the other may produce tailored software for a particular customer while the third creates packaged software. If company's business is in producing software it is very likely that the organizational structure already reflects this, for example, packaged software company is likely to divide its units according to its product families while tailored software producer may group its functions according to the customers lines of business.

The third step is to create the hierarchy of sub-domains. Here we combine the organizational structure and the nature of the software development. In this way we produce HDM for the business aspect. The resulting hierarchy is not by definition replica of the company's organization chart because we only want to model the applications that are used and produced by the organization. There is no point to model such an organizational division where there are different departments to market product to east and west side of a country, i.e., we cannot see much point in producing west side of the country software apart from the east side of it.

The fourth and last step of the first phase creates hierarchy for the software aspect. This is based mainly to the existing ways to produce software intensive solutions. Often the basic division to middleware and system software is sufficient (cf. Jacobson et al., 1997).

As a result of the first phase we have the initial HDM where both the business and software aspects are by and large defined (see Figure 5.3). In the HDM higher levels in the hierarchy mean more general not more abstract levels. Each of the identified sub-domains may be further refined to include more specific sub-domains in.

The Information Technology Research Institue (ITRI) is an organizatoin that carries out research companies. Research is done in projects. ITRI has also some educational services that it offers. Now ITRI is designing software to support its project planning. This development project decided to use HDA approach. The first task is to create initial hierarchy for the domain. ITRI is creating software only for itself and its organizational structure is functionally grouped, thus hierarchy for the business aspect is done according to these functions. Because the target of the development is project planning software, the project function is examined more closely. The software aspect hierarchy is created according to middleware and system software layers.

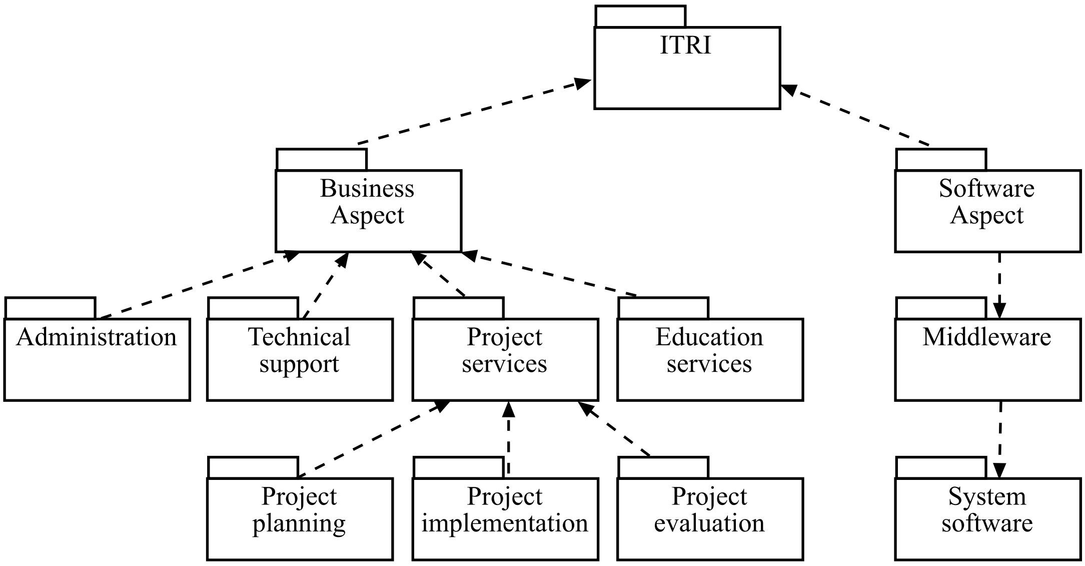


FIGURE 5.3 Example of the Hierarchical Domain Model.

5.4.2 Define Sub-Domain

As stated earlier, one of the hardest tasks is to define boundaries for domain and this is the aim of the second phase of HDA. The second phase has three steps:

1. Define the boundaries of the current development project
2. Find information sources for the sub-domain
3. Specify the problems and requirements in the sub-domain

In the first step we use the created HDM as a starting point and try to figure out where in the sub-domains the current development is. Boundary setting should be easier because we can see which business areas are outside of the sub-domain and which are inside. The second step finds possible information sources. Finding out who are the domain experts can start this. The selected experts can point out more relevant information sources in the company. Also, we have to scan the literature and perhaps study the similar software products in the market.

The third step specifies the problems and requirements that are particular in the identified sub-domain. We already have the overall picture of the software project's objectives because otherwise we could not been able to do the first step in this phase at all. In this step we try to figure out the similarities that are between the programs in the sub-domain and try to define also the variation points that are needed. As a result of the second phase we have defined our sub-domain and perhaps refined the overall HDM in this sub-domain area (cf. Figure 5.3).

5.4.3 Analyze Business Aspect

In this phase domain analyzers concentrate on the identified sub-domain and try to find reusable information inside of it. When we refine the sub-domain we are looking for the services that the software in the sub-domain must have in order to be useful for its users (cf. Jaaksi et al., 1999; Jacobson et al., 1999). The third phase is comprised from the following steps:

1. Define the business processes
2. Define the key concepts
3. Define the needed services
4. Analyze the identified services
5. Classify the services (first according to Common Process and second, locate services into hierarchy)

The first step defines business processes. After we have defined the sub-domain of interest we gather more information about the problem area. In order to understand the work in the sub-domain we have to create functional descriptions of it. This can be done with UML's activity diagrams.

The second step defines sub-domain's key concepts. After the domain analyzers possess understanding about of what kind of business processes are executed in the sub-domain they can identify the key concepts in it. Identified concepts are described with the class diagram. There is no point in identifying definite set of attributes for the classes and attributes should be described only if there is a danger that the attributes and concepts (classes) can be mixed. In any case the operations for classes are not described.

In the third step the analyzers define the services that the software must offer for the user. When analyzers are dealing with the business aspect of the sub-domain they should try to find services that users of the software need in order to accomplish their every day tasks (see Jacobson et al., 1999). Users of the software are not interested how particular service is implemented but rather what is the gain to them using the implemented software.

The services are described with packages (UML). The service packages has following characteristics, among other things, (Jacobson et al., 1999):

1. it contains a set of functionally related classes,
2. it is indivisible,
3. it often has very limited dependencies toward other service packages,
4. the functionality defined by a service package can, when designed and implemented, be managed as a separate delivery unit,
5. they may be mutually exclusive, or they may represent different aspects or variants of the same service, and
6. they constitute an essential input to subsequent design and implementation activities, in that they will help structure the design and implementation models.

The fourth step analyzes the found services. In this step the analyzers identify relationships between service packages and their functionality. Each service package is analyzed and its core functionality, i.e., the functionality that does not vary, is separated from the modifiable functionality. These help create adaptation points in the later use of these service packages (i.e., components). Relationships between service packages help define also the service package's interfaces.

The fifth step classifies the services. As stated earlier, the classification makes the difference between domain analysis and development method's analysis and for this reason we also emphasize the classification step. In HDA classification involves two stages. The first classification stage is done according to

Common Process, that is: cluster descriptions, abstract descriptions, classify descriptions, generalize descriptions and construct vocabulary. The aim is to find service packages the user of the software needs. Basically this means that identified key concepts (classes) are located into preferably one package.

In the second classification stage found service packages are located into the hierarchy of sub-domains. This is the most important task in the HDA and this makes its hierarchy useful to later development projects. In this stage services that can be recognized as useful in other sub-domains are located into the hierarchy, that is, if we can see that certain kind of service can be used in the whole company we locate this service to the company level in the hierarchy. If we see that the service is useful, for example, in the development of project related software (see Figure 5.4) we locate the service in the project level. The domain analyzer finds the appropriate sub-domain level from the hierarchy by asking himself (a) where these kinds of services are needed, and (b) is it possible to create a more general service that could be used more widely inside the organization. According to the answer, the analyzer locates the service package inside the appropriate level of the hierarchy. This way we try to predict where the identified service can be (re-) used, thus making it easier to find.

After the third phase we have defined within the specified sub-domain the service packages and requirements for the application. HDM is filled with the found service packages so that each component is put into the hierarchical level where it is potentially seen as (re-) usable (see figure 5.4). The identified domain experts should validate the results.

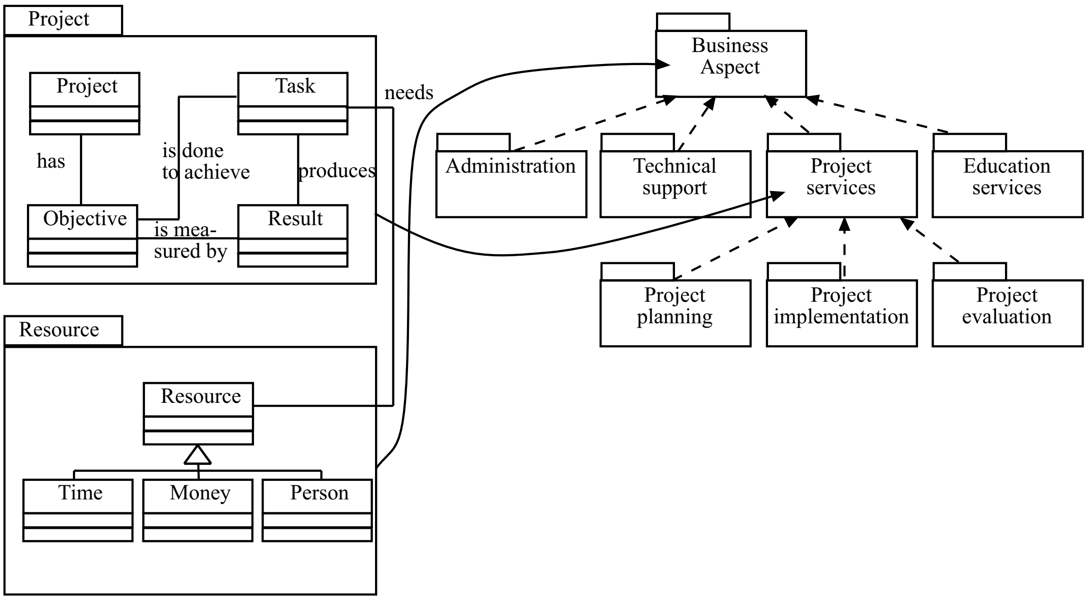


FIGURE 5.4 Example How to Locate Found Services into the HDM.

5.4.4 Analyze Software Aspect

In the fourth phase the software aspect of the HDM is refined. Here we try to find the functionality that the existing software systems offer for the basis of the solution. It is quite usual that this phase is straightforward and it should not create any noticeable difficulties. The domain experts in this phase are those people inside the organization who are familiar with the existing technical solutions and they can say what might happen to the technology in the future. These people are most often system designers and programmers. The steps two to five of Common Process are used in this phase although they are not explicitly shown here. The fourth phase has three steps:

1. Define the layers for the software aspect
2. Define the used software solutions
3. Classify the found solutions

The structure of the software aspect can be based on the layered architecture (cf. Jacobson et al., 1997) where middleware and system software are the lowest layers. The middleware layer is independent of the operating system and examples of the middleware components are interfaces to DBMS, graphical user interface toolkits, and object request brokers. Java as a programming language gives the same kind of independence and is a good way to create a corporate wide standard for inter-process communication. The system software layer includes such components as an operating system, data communication protocols, and device drivers.

It is not always that easy to draw the line between the middleware and system software components (Jacobson et al., 1997) but in many cases it is obvious. If different opinions about the location of a specific component occur, the criteria of operating system independence should be used, i.e., if the component is operating system dependent it is a system software component, otherwise it is a middleware component. After the fourth phase we have a completed HDA. The results from the fourth phase serve as a basis for constraint identification. We also have an extensive set of requirements and constraints that the domain's software has to follow. The identified domain experts should validate the results.

5.4.5 Use Hierarchical Domain Analysis Results

Although we call this the fifth phase of HDA it is not exactly a phase. Here we present how results of the HDA can be used to foster company wide reuse and how they contribute to every day software development. First of all, the hierarchical sub-domain structure eases the phase of defining sub-domains. When we have created company-wide HDM we can use this model to identify where our software exactly belongs, that is, we can identify the sub-domain into which our software belongs. Furthermore, we can easily determine what is outside of a particular sub-domain. Thirdly, because of the second stage classification (i.e., locating the found services or components into the hierarchy of sub-domains) we can find the services and components that were created in other sub-domains. Moving from the sub-domain at hand upwards the hierarchy can do this. In this way we can find services that other domain analyzers have seen as useful in other domains. Of course this does not imply that every general service identified is always useful. The fourth way to use results is to classify the services. As a matter of fact this classification is used when we scan through the hierarchy for the available services.

HDA results can also be used to define the software architectures. Many other authors have also presented that DA results can be used this way (e.g., Tracz, 1995) and HDA does not differ in this sense. In the following we present three steps to create software architecture but on the contrary to many other DA methods that produce software architectures, we see that this has to be done every time when new software is created. This means that we are not aiming to produce any kind of reference architecture. The first step to define software architecture creates layered component architecture. Jacobson et al. (1997) have defined a layered component architecture that can be used to separate service packages and software components to different layers. These layers are application, component, middleware, and system software layer.

The second step defines the potential solutions. It is not feasible to determine all the possible solution alternatives, thus only the most interesting ones should be explored. What makes a solution interesting depends on company's existing solutions, technologies and, of course, experts' judgment.

The third step defines software architecture. The HDM can serve as a basis for the design of the software architecture. The software architecture shows the architecturally significant components and their connections (Shaw and Garlan, 1996). Kruchten (1995) argues that the software architecture should be modeled from multiple viewpoints and suggests the logical, process, implementation, deployment, and scenario views. The scenarios should be based on the identified services (cf. Jaaksi et al., 1999; Jacobson et al., 1999).

Each one of the views provides different point of view to the structure of a system (Clements and Northrop, 1996). The logical view defines the logical structure of the software. It gives overall structure of the software and shows main parts of it and defines the main solutions for it. Logical structure description can be based on the layered component architecture. The logical view model can be based on the service packages that were identified during the analyzing the business aspect phase. In the development view the components and their relations are described explicitly. This view should also serve as basis for the separation of the development units. Here the whole HDM should be scanned to get information but the resulting development view needs to be further refined. The software developers use the development view and it can be described with the package and class diagrams.

The process view illustrates what components belong to the system and how these components communicate with each other. This can be described with the component diagram. The reader should note that components could not communicate by any other means than what is defined in the software aspect of the domain model. The components have to use always some existing means to communicate, for example, via middleware or by some programming language specific feature (e.g., Java RMI).

The physical view presents how the components are located to the different machines and processors. The deployment diagram can describe this. This view is quite software specific and it cannot be aided by the results of the domain analysis.

After the third step of the software architecture modeling we have quite complete logical and process views and drafts for the development and physical views (see figure 5.5). These models are further refined by the software development method's own architectural design phase.

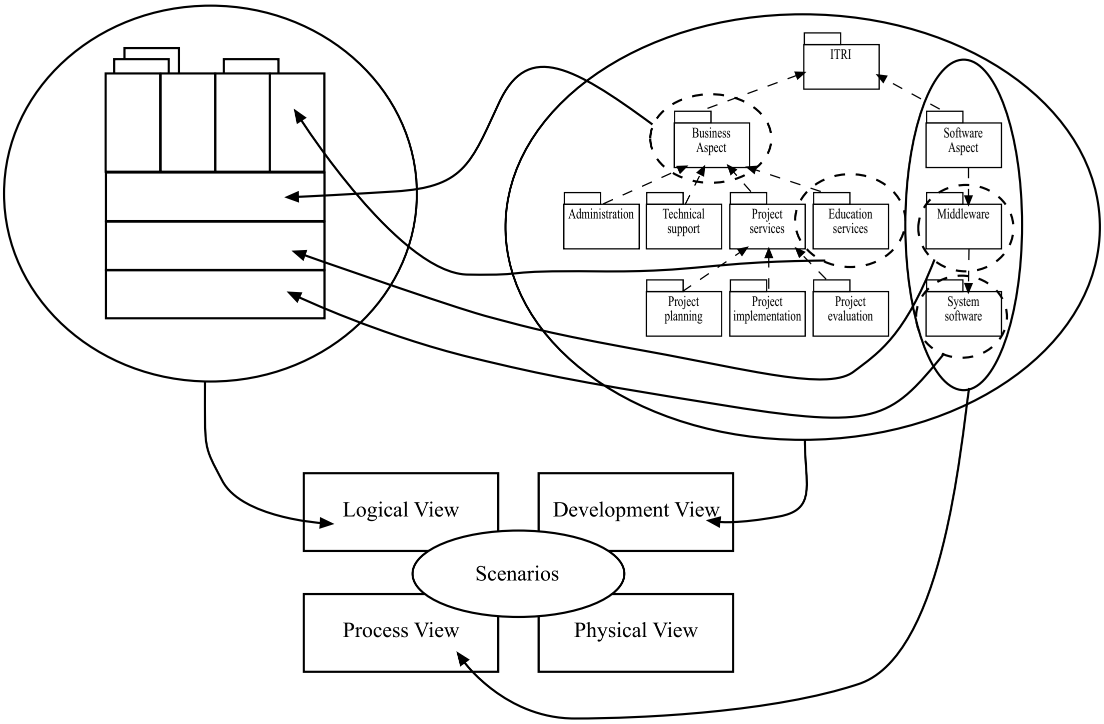


FIGURE 5.5 A Sketch How to Use Hierarchical Domain Analysis Results in the Creation of the Software Architecture.

5.5 Conclusions

We started this study by asking: "How should the domain(s) be analyzed so that all solutions readily available could be part of the resulting software solution, and that results can be produced and used in everyday software development?" We suggest that the answer to the first part of the question could be: by adding a phase to model the whole company as a hierarchy of sub-domains and by using this hierarchy to preserve reusable information at the level where we can see the reuse opportunities for it. In hierarchical domain analysis (HDA) the most important task is the classification of the components into the hierarchy of sub-domains. This way the (re-) use of the components can happen in the other sub-domains without experience in, or a priori knowledge about it.

We propose that the answer to the latter part of the question could be that by using HDA as part of the software development (see Prieto-Díaz, 1987). HDA produces a hierarchical domain model (HDM) that is used and refined during every software development project. This makes a HDM dynamic. It evolves with the environment and it allows HDM to be refined when experience over some sub-domain grows. Furthermore, we added a phase to HDA where the results of HDA are used during the software development project.

HDA is not biased towards any particular implementation technology (e.g., object-oriented) or method (e.g.,Unified Process or SA/SD) apart from the fact that we have used UML to create our models. A HDM can be used in the organizations that want to apply domain analysis to facilitate the reuse. The initial experiences have been positive from the companies that have utilized our HDA approach.

There are still many open questions in HDA approach. The drawbacks of HDA are that it does not contribute to the documentation of the components and it also neglects the testing of the components. Also, UML-based notation may be improved when empirical evidence conserving its suitability is at hand. One quite fruitful area of research would be to define the different views for domain analysis (cf. Kruchten, 1995 and software architectures). Nevertheless, we think that a HDM is useful in the classification of the components, although it was not shown explicitly here. We see these aspects very important and further research is still needed in these areas.

Acknowledgements

Tekes Technology Development Centre, Finland and companies participating in the PISKO project supported this research. Veikko Halttunen, Tero Päivärinta, and Jarmo Ahonen from the Information Technology Research Institute at the University of Jyväskylä, and Jaroslaw Skwarek and Ari Hirvonen from the TietoEnator Corporation gave number of valuable comments during earlier drafts of this paper. I would also like to thank the anonymous reviewers whose comments helped improve the focus and clarity of this paper.

References

Arango, G. ”Domain analysis - From Art Form to Engineering Discipline,” Proceedings of the 5th International Workshop on Software Specifications and Designs, 1989, pp. 152-159.

Arango, G. ”Domain analysis methods,” in (eds.) Schäfer, W., Prieto-Díaz, R., Matsumoto, M. Software Reusability, Ellis Horwood Ltd., 1994, pp. 17-49.

Arango, G., Prieto-Díaz, R. ”Domain Analysis Concepts and Research Directions,” in (eds.) Prieto-Díaz, R., Arango, G. Domain Analysis and Software Systems Modeling. IEEE Computer Society Press Tutorial, 1991, pp. 9-32.

Bayer, J., Flege, O., Knauber, P., Lagua, R., Muthig, D., Schmid, K., Widen, T., DeBaud, J.-M. ”PuLSE: A Methodology to Develop Software Product Lines,” Proceedings of the 1999 Symposium on Software Reusability, pp. 122-131.

Biggerstaff, T., Richter, C. ”Reusability Framework, Assessment, and Directions,” IEEE Software, March 1987, pp. 41-49.

Booch, G., Rumbaugh, J., Jacobson, I. The Unified Modeling Language User Guide, Addison Wesley Longman Inc., 1999.

Clements, P., Northrop, L. ”Software Architecture: An Executive Overview,” Technical Report, CMU/SEI-96-TR-003, Software Engineering Institute, Carnegie Mellon University, 1996.

Codenie, W., De Hondt, K., Steyaert, P., Vercammen, A. ”From Custom Applications to Domain-Specific Frameworks,” Communications of the ACM, Vol. 40, No. 10, October 1997, pp. 70-77.

D’Souza, D., Wills, A. Objects, Components, and Frameworks with UML: The Catalysis Approach, Addison-Wesley, 1999.

Forsell, M., Halttunen, V., Ahonen, J. ”Use and Identification of Components in Component-based Software Development Methods,” To appear in the Proceedings of the Sixth International Conference on the Software Reusability, Vienna, Austria, June 27-29, 2000.

Freeman, P. ”Reusable Software Engineering: Concepts and Research Directions,” ITT Proceedings of the Workshop on reusability in Programming, 1983, pp. 129-137.

Horowitz, E., Munson, J. ”An Expansive View of Reusable Software,” IEEE Transactions on Software Engineering, Vol. 10, No. 5 September 1984, pp. 477-487.

Jaaksi, A., Aalto, J.-M., Aalto, A., Vättö, K. Tried & True Object Development: Industry-Proven Approaches with UML, Cambridge University Press, Cambridge, 1999.

Jacobson, I., Griss, M., Jonsson, P. Software Reuse: Architecture, Process and Organization for Business Success, ACM Press, New York, 1997.

Jacobson, I., Booch, G., Rumbaugh, J. The Unified Software Development Process, Addison Wesley Longman, Inc., 1999.

Jackson, M. ”Problems, Methods and Specialisation,” Software Engineering Journal, Vol. 9, No. 6, November 1994, pp. 249-255.

Jarzabek, S. ”Modeling Multiple Domains in Software Reuse,” Proceedings of the 1997 Symposium on Software Reusability, 1997, pp. 65-74.

Karlsson, E. (ed.) Software Reuse: A Holistic Approach, John Wiley & Sons Ltd., Chichester, 1995.

Krueger, C. ”Software Reuse,” ACM Computing Surveys, Vol. 24, No. 2, June 1992, pp. 131-183.

Kruchten, P. ”The 4+1 View Model of Architecture,” IEEE Software, Vol. 12, No. 6, November 1995, pp. 42 - 50.

Lam, W., McDermid, J. ”A Summary of Domain Analysis Experience by Way of Heuristics,” Proceedings of the 1997 Symposium on Software Reusability, 1997, pp. 54-64.

McCain, R. ”Reusable Software Component Construction: A Product-Oriented Paradigm,” Proceedings of the 5th AIAA/ACM/NASA/IEEE Computers in Aerospace, 1985, pp. 125-135.

McIlroy, M. ”Mass-produced Software Components,” in (eds.) Buxton, J.M., Naur, P., Randell, B. Software Engineering Concepts and Techniques, 1968 NATO conference on Software Engineering, Petrocelli/Charter, Belgium, 1976, pp. 88-89.

Mintzberg, H. Structure in Fives: Designing Effective Organizations, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1983.

Mintzberg, H. ”A Typology of Organizational Structure,” in Miller, D., Friesen, P. Organizations A Quantum View, Prentice-Hall Inc., 1984, pp. 68 - 86.

Moore, J., Bailin, S. ”Domain Analysis: Framework for Reuse,” in (eds.) Prieto-Díaz, R., Arango, G. Domain Analysis and Software Systems Modeling, IEEE Computer Society Press, 1991, pp. 179-203.

Neighbors, J. ”Software Construction Using Components,” Ph.D. Thesis, Department of Information and Computer Science, University of California, Irvine, 1980.

Neighbors, J. ”DRACO: A Method for Engineering Reusable Software Systems,” in (eds.) Biggerstaff, T., Perlis, A. Software Reusability, Volume I, Concepts and Models, ACM Press1989, pp. 295-319.

Parnas, D. ″On the Design and Development of Program Families,″ IEEE Transactions on Software Engineering, Vol. SE-2, No. 1, March 1976, pp. 1-9.

Prieto-Díaz, R. ″Domain Analysis for Reuse,″ Proceedings of COMPSAC ′87, 1987, pp. 23-29.

Prieto-Díaz, R. ″Domain Analysis: An Introduction,″ Software Engineering Notes, Vol. 15, No. 2, April 1990, pp. 47-54.

Prieto-Díaz, R., Arango, G. (eds.) Domain Analysis and Software Systems Modeling. IEEE Computer Society Press Tutorial, 1991.

Prieto-Díaz, R. ″Historical Overview,″ in (eds.) Schäfer, W., Prieto-Díaz, R., Matsumoto, M. Software Reusability, Ellis Horwood Limited, 1994, pp.1-16.

Shaw, M., Garlan, D. Software Architecture: Perspectives on an Emerging Discipline, Prentice-Hall Inc., New Jersey, 1996.

Simos, M. ″Organization Domain Modeling (ODM) Guidebook, Version 2.0,″ Informal Technical Report for Software Technology for Adaptable, Reliable Systems (STARS), STARS-VC-A025/001/00, June 14, 1996.

Tracz, W. Confessions of a Used Program Salesman: Institutionalizing Software Reuse, Addison-Wesley Publishing Company, 1995.

Villarreal, E., Batory, D. ″Rosetta: A Generator of Data Language Compilers,″ Proceedings of the 1997 Symposium on Software Reusability, Boston, United States, May 17 - 20 1997, pp.146-156

Wartik, S., Prieto-Díaz, R. ″Criteria for Comparing Reuse-Oriented Domain Analysis Approaches,″ International Journal of Software Engineering and Knowledge Engineering, Vol. 2, No. 3, September 1992, pp. 403 - 433.

Wegner, P. ″Varieties of Reusability,″ ITT Proceedings of the Workshop on Reusability in Programming, 1983, pp. 30-44.

Whittle, B. ″Reusing requirement specifications: Lessons Learnt,″ Proceedings of the 7th Annual Workshop on Software Reuse, Andersen Consulting Center in St. Charles, Illinois, August 28-30 1995.

6 A MODEL FOR DOCUMENTING REUSABLE SOFTWARE COMPONENTS

Forsell, M.[†], Päivärinta, T.[‡], "A Model for Documenting Reusable Software Components", Re-submitted for publication to Database for Advances in Information System's special issue on Component Based Development. Copyright may be transferred without further notice and the accepted version may be posted by the publisher.

[†]Chydenius Institute, University of Jyväskylä, Finland.
[‡]Agder University College, Department of Information Systems, Norway.

Abstract

Effective reuse of software components requires effective means for documenting and communicating several kinds of related information among many stakeholders in the reuse process of those components. The existing models for component documentation pay little attention to the issue of how the documents are processed and communicated among the stakeholders during the reuse process. We address the need and sketch an abstract model for an explicated genre system for this component documentation. Our model highlights the communication viewpoint to component documentation, elaborating on the traditional models that have mainly provided suggestions for the documentation's content. We give an example of how the model was used in defining documentation for a reusable component in a software company. The model provides a theoretically informed basis for research on documentation practices related to the component-based software industry. It can be used as a starting point to develop and implement organization-specific component documentation and to enhance the related communicative practices in a software company.

Categories and Subject Descriptors: D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement – *documentation*; D.2.13 [**Software Engineering**]: Reusable Software – *reusable libraries, reuse models*; K.6.3. [**Management**

of Computing and Information Systems]: Software Management – *software development, software process*

General Terms: Design, Documentation, Management, Standardization, Theory

Additional Key Words and Phrases: Component documentation, genre theory, genre system, reuse process, software component

6.1 Introduction

In the 1960's, McIlroy (1968) introduced the idea of software development based on code components. Currently, the notion of a software component may include, apart from plain code, other artifacts of the software development process as well: for instance, requirements, design documents, test scripts, and user manuals (e.g., Freeman, 1983; Krueger, 1992; Lim, 1997). The whole idea of component-based software development (CBD) thus pursues an efficient and effective reuse process; i.e., the activities of managing, producing, brokering, and consuming all kinds of software components (Lim, 1997; STARS, 1992), beyond any single software development process at hand (Figure 6.1).

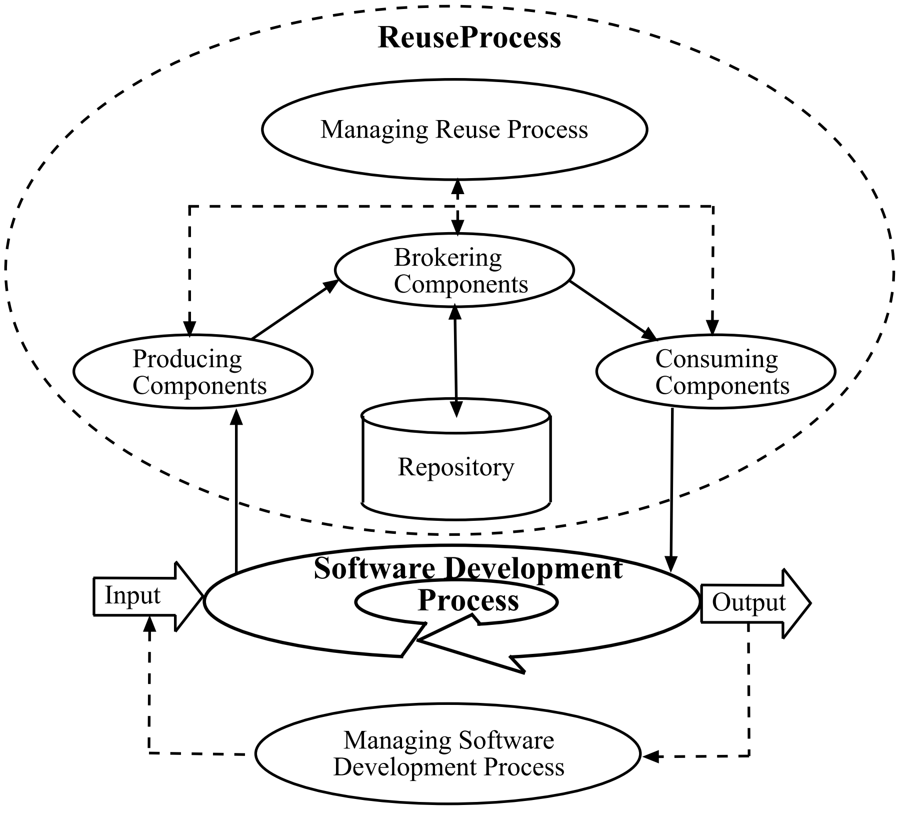


FIGURE 6.1 Model for reuse-oriented software development. In the model essential features of reuse processes are tied up to the software development process.

In the reuse process (Lim, 1997; STARS, 1992), the managing tasks set general goals and rules for the reuse support. The production of a component is rather similar to an ordinary software development project. However, some extra effort is involved, for example, in testing the component as such and creating documentation for potential reusers of the component. In connection to production, domain analysis is needed for defining components (Arango, 1989, Lam & McDermid, 1997, Prieto-Díaz, 1994). Brokering includes procedures for selecting appropriate components to a repository, certifying the need for the components, and assuring their quality. Brokering information should convince the reusers that a component is relevant for the declared purpose in the first place, and that it can be trusted. Consuming takes place, on the one hand, when reuse-oriented developers are searching for, trying to understand, and integrating a component in a software product. The reused component may be either black-box or white-box component because in both cases all these steps appear to be necessary, for example in the adaptation phase we may alter the implementation of a white-box component but to a black-box component we may alter the functionality only by parameters. On the other hand, components are also consumed in the subsequent phases of the development project. For instance, besides the use of a design component in the design phase, it will be used also in the implementation and testing phases.

A reusable component thus should be made usable in multiple software implementations and development processes along time. If we agree on Lim's (1997) conceptualization of the reuse process above, the following differences between conventional software development and CBD can be identified: a component must be made reusable in different projects from that for which it was created, a creator of a component is often a different person or development group than its potential user, and the creator may reside in a different location geographically and/or work for different organization. The idea of CBD thus includes a number of special temporal, geographical, logical, and organizational characteristics and challenges if compared to the conventional software development process.

These challenges suggest that a software component must include documentation of its relevant characteristics for reuse, since one cannot assume that the original producers and potential reusers of a particular component can (or should) communicate directly with each other in all circumstances. For instance, the geographical, temporal, or organizational boundaries may prevent direct communication. The development process based on the reuse of software components thus involves an interesting communicative process in which documents play a significant role. This paper addresses the viewpoint of organizational communication as a means for elaborating the contemporary models of this documentation, hereinafter referred to as component documentation. Throughout the reuse process, component documentation involves several stakeholder roles that can be assigned to individuals: the producer(s) of the component (and documentation), the user(s) of the component, and the brokers and managers who facilitate the transmission of the component from the production activity to software development efforts that consume it. Accordingly, component documentation should support varying communication needs. Although the literature has suggested a few models for component documentation (Prieto-Díaz and Freeman, 1987; NATO, 1992; Karlsson, 1995; Basili and Abd-El-Hafiz, 1996; Sametinger, 1997), it has generally been a neglected area of research (Dusink and Katwijk, 1995). The existing models do not explicitly build on any theoretical base that would consider the communicative or organizational aspects of component documentation. Rather, they are mainly based on ad hoc recommendations of information content to be included. Our contribution to research and practice is twofold. Firstly, this paper elaborates from the existing models for component documentation by addressing the communication viewpoint and organization context with regard to component documentation. Secondly, these viewpoints are illustrated further with an abstract-level model of component documentation. Our model draws on Lim's (1997) model of the reuse process adding the theoretical viewpoint of genre of organizational communication (Yates & Orlikowski 1992) and especially the related concept of a genre system (Bazerman, 1994, Orlikowski & Yates 1998). Recently, Spinuzzi & Zachry (2000) have suggested genre theory as a basis for designing software documentation intended for the end user. Our paper denotes the communicative role of documentation in the reuse process for software developers.

The next section introduces genre theory, with a justification to use it for the analysis of component documentation and a genre-system-based framework for doing so. Section three scrutinizes three previous models for component documentation according to the framework. The elaborated model is presented in section four with an example that illustrates its use in a software-producing organization. Section five discusses implications of the model on research and practice. Section six concludes with suggestions for further research.

6.2 Genre Theory: A Basis for Analyzing Component Documentation

The notion of genre originates in the Greek word genos meaning 'race', 'kind', 'sort', or 'class' (Zimmerman 1994). The concept has been used through centuries for analyzing pieces of literature and rhetorical utterances (Bakhtin 1952/53). In the 1950's, Bakhtin (1952/53) introduced the concept of speech genre, which he considered useful for analyzing everyday communication, e.g. the documents used in contemporary organizations. More recently, Yates and Orlikowski (1992, Orlikowski & Yates 1994, Yates, Orlikowski & Okamura 1999), drawing on Miller's (1984) seminal discussion on rhetorical genre, elaborated the concept of genre of organizational communication. They defined it as a typified and recurrent communicative action characterized primarily by its substance in an organizational context and, to some extent, by its form. Typically, a genre is explicitly identified and enacted as an established way to communicate within a particular community (Yates & Orlikowski 1992). A group of stakeholders involved in a CBD process (including the reuse process) could be regarded as such a community.

The genre theory of organizational communication was chosen as it complements the process-based viewpoint to CBD, from which we started to conceptualize our research in the first place. Bazerman (1994) denoted that particular genres may interrelate in recurrent communicative settings forming systems of genre or, as simply expressed by Yates, Orlikowski, and Rennecker (1997), genre systems. The concept of genre system thus corresponds directly to the idea of process (Conger & Schultze 1999): both of them focus on recurrent actions in organizational context, the former from the viewpoint of communication patterns and the latter from the viewpoint of meaningful tasks that are interrelated. Hence, if one can speak of the generic reuse process, as Lim (1997) does, one should be also capable of conceptualizing a generic genre system of communication needed in the related tasks as well. As soon as a genre system is identified, it becomes possible for the stakeholders to discuss further about desired socio-organizational practices and norms for that communication; concerning the genres involved and their actual technical implementations (Brown & Duguid, 1994, Päivärinta, 2001). Yates & Orlikowski (1992) note that genres can be analyzed and depicted at various levels of abstraction. Alike Lim's abstract and generic model of the reuse process can be operationalized in organizations in varying detail, one should thus be as well able to discuss about generic aspects of a communicative genre system in the reuse process and to specify it further in any organizational context. The discussion could proceed until an actual implementation of a communication system and repository implementation for CBD, as necessary.

Moreover, the notion of genre system provides a more general-level basis for the generic discussion for our purposes than the definitions of single genres would give. As the field of component documentation is a novel and evolving one, it makes sense first to start with requirements for a genre system in total, which could be later on specified towards more detailed genres related to particular organizational settings. Representing a start for an effort of this kind, this article will stay mostly on the level of discussing this genre system as a whole instead of going into detailed definitions of individual genres. Orlikowski and Yates (1998) have elaborated a pragmatic framework for analyzing genre systems according to six aspects: why, what, how, who/m, when, and where (Table 6.1). We will use this framework in the subsequent analysis of existing models for component documentation as well as in our effort to elaborate them.

Among the myriad of theories on organizational communication, few options to scrutinize communication itself with regard to organizational processes exist. Among those, the theories drawing on the concept of speech act (Auramäki & Lyytinen, 1996), originating in Austin's (1962) work on linguistics, have been used for analysing and creating documentation systems (Auramäki, Lehtinen & Lyytinen, 1988; Gordon & Moore, 1999). However, speech acts deal most essentially with an analysis of particular instances of documents and the effects intended by the producer and received by the perceiver of that particular doc-

TABLE 6.1 A Framework for Analyzing and Describing Genre Systems. (Orlikowski & Yates 1998)

Why?	Declares the ultimate purpose of the whole genre system in question and the purposes of the genres that constitute it. For instance, the genre system of "scheduling documentation in the project" aims at effective timing and organization of project tasks. Among these documents, the "Gantt-chart" carries a specific purpose for providing a rough overview of the whole project, etc.
What?	Declares the expectations about the information content of the whole genre system, and the sequence and information content of the genres that constitute the whole.
How?	Declares what is expected from the form of the genre system as well as from each of the constituent genres; including communication media, ways to structure information within the genre system, and linguistic features specific both for the genre system and for its particular genres.
Who/m?	Declares the stakeholders using the genre system and its constituent genres for communication; i.e., the primary and secondary roles of producers and users of that information.
When?	Declares the temporal expectations on the genre system and the particular genres. For instance, specified time periods to read and comment on a documented genre, or specified deadlines to produce instances of a particular document genre.
Where?	Declares the expectations on locations and places where the genre system in question and its constituent genres should appear physically and logically.

ument, instead of serving our purposes for a type-level modeling of an area of interest. Typical occurrences of diverging speech acts could, however, be analyzed from a set of identified genres in a domain with regard to identified stakeholders (Päivärinta, 2001). The concepts of traditional data modeling, (related to component documentation), could also be identified relevant with our discussion. However, they focus mostly on the actual implementation of the information content in question; for instance, the concepts entity or object (of component documentation) do not necessarily capture the context of a communicative situation in which a piece of documentation is produced or used. Again, one could well proceed towards the technical entities and objects of component documentation from a genre-based scrutiny of the documentation's meaningfulness in an organizational context.

6.3 Previous Models for Component Documentation in Light of the Genre-Based Framework

Dusink and Katwijk (1995), in their survey on the reuse literature, argued that component documentation was a neglected area of research. However, a few models for component documentation have been published. We chose the following ones for more detailed analysis: Sametinger's (1997), Karlsson's (1995), and the NATO model (1992). We naturally recognize a wide base of literature concerning the documentation of code components (Frakes & Pole, 1994; Henninger, 1997). However, the emphasis in these is on finding and retrieving components, or they address other narrow and specific areas: e.g., how to document loops (Basili & Abd-El-Hafiz, 1996). We considered the three models mentioned earlier appropriate for enlightening the state-of-art in general and for fulfilling the purposes of this paper. We analyzed and compared Sametinger's, Karlsson's and NATO's models in light of Orlikowski's and Yates' (1998) genre system framework. Appendix 1 presents a detailed comparison of the models. This section attempts to summarize these models and their shortcomings as a basis for our model to be elaborated in section 4.

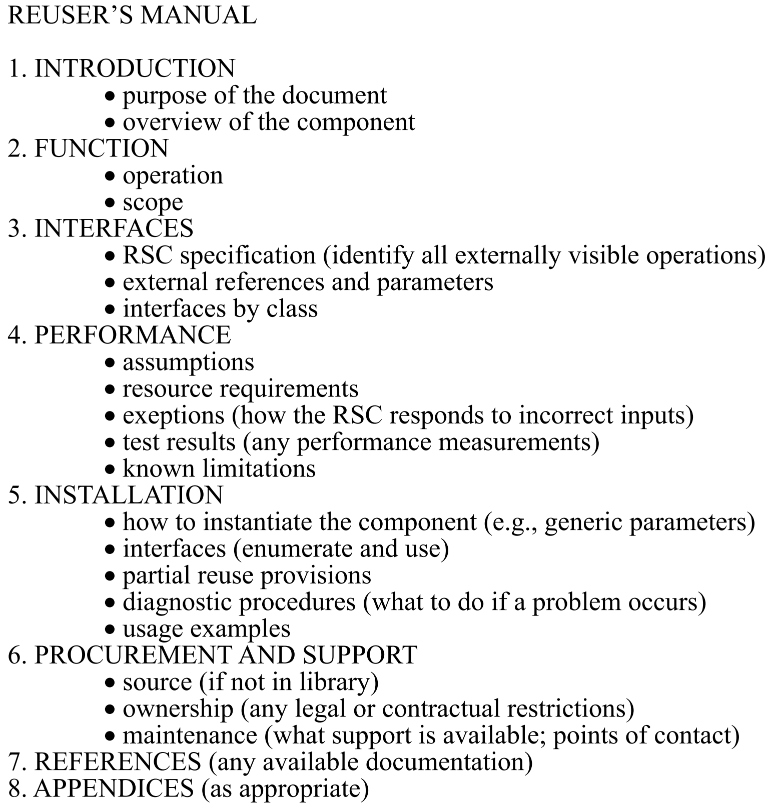
6.3.1 The NATO Model for the Reusable Software Component Documentation (NATO, 1992)

The NATO standard for the development of reusable software components (RSC) pursues maximum potential for software reuse. Documentation of a reusable component provides a key part of its reuse value, playing a dual role. First of all, it must conform to the needs of the immediate software system under preparation. Secondly, it must give explicit guidance for reusers. A reuser must be able to access quickly the information s/he needs.

Documentation must comply with accepted standards of the user community, being consistent in organization and in format, and reflecting changes in the code. Documentation should be self-contained and possibly accompanied with the reusable component. Finally, documentation should be in machine-readable form and understandable by others.

Reuse library emerges as a special concern. Documentation must support the classification, identification and retrieval of components. A component's functionality should be easily viewable through abstracted summaries. The dependencies must be explicitly described and there should be classification information.

Also the assessment of RSC should be described. Different kinds of metrics about the reusability and quality should be stated as well as known problems and recommended enhancements. The potential reuser should also be aware of any commercial or legal restrictions, and how to access the component if it is not physically in the reuse library.



REUSER'S MANUAL

1. INTRODUCTION
   - purpose of the document
   - overview of the component
2. FUNCTION
   - operation
   - scope
3. INTERFACES
   - RSC specification (identify all externally visible operations)
   - external references and parameters
   - interfaces by class
4. PERFORMANCE
   - assumptions
   - resource requirements
   - exeptions (how the RSC responds to incorrect inputs)
   - test results (any performance measurements)
   - known limitations
5. INSTALLATION
   - how to instantiate the component (e.g., generic parameters)
   - interfaces (enumerate and use)
   - partial reuse provisions
   - diagnostic procedures (what to do if a problem occurs)
   - usage examples
6. PROCUREMENT AND SUPPORT
   - source (if not in library)
   - ownership (any legal or contractual restrictions)
   - maintenance (what support is available; points of contact)
7. REFERENCES (any available documentation)
8. APPENDICES (as appropriate)

FIGURE 6.2 Information content of the NATO reuser's manual. (1993, p. 8-5)

Normal documentation does not fully meet the needs of the RSC reuser; additional support should be provided in a reuser's manual for every component (Figure 6.2). The manual should follow a standard format.

6.3.2 The REBOOT Component Model (Karlsson, 1995)

According to Karlsson, a reusable component is a part of a product at some level of development (requirements, design, code), together with information about the component to make reuse feasible. The reusable component must be self-contained. Hence, when a company has decided on which components to reuse, a decision on how these components should be packaged for reuse must follow. The component model describes the information needed for a packaged reusable component.

Karlsson provides an entity-relationship-based description of his component model (Figure 6.3). With small components only parts of the model may be regarded as useful.

The classification information aids component identification and retrieval. The component qualification information describes the quality and reusability of the component. Information is used while deciding whether the candidate component fulfils requirements for quality and reusability. This information also

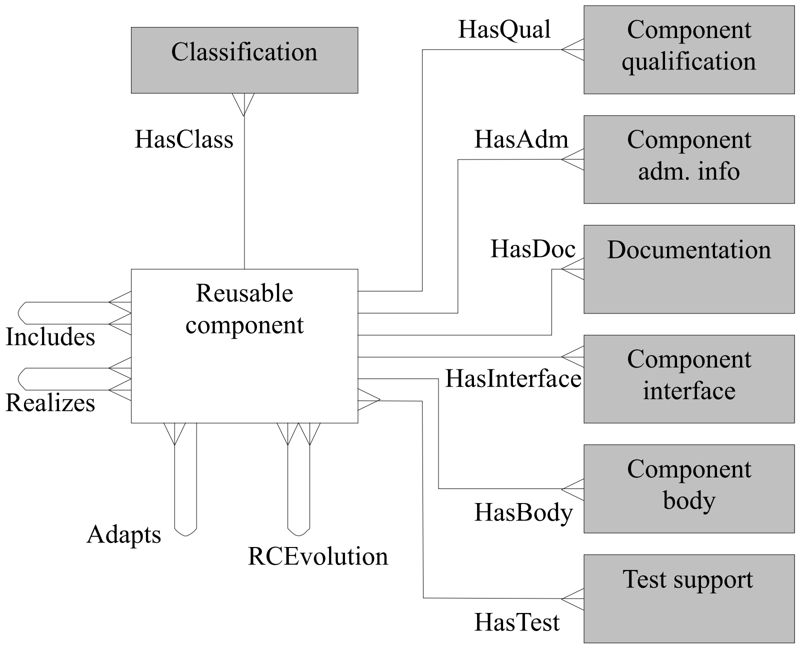
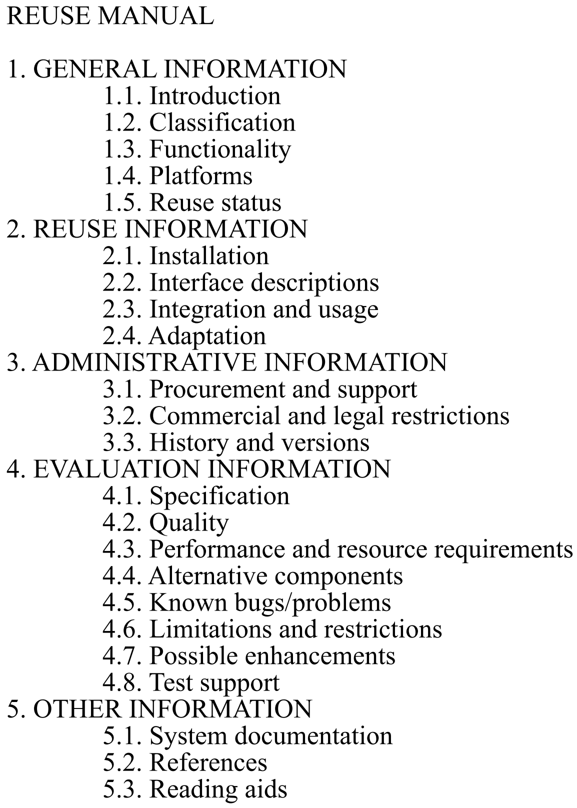


FIGURE 6.3 The REBOOT component (documentation) model. (Karlsson, 1995, p. 82)

tracks the reuse history of the component, i.e., the experiences and problems. The component administrative information includes general information about the component, including authorization and prizing. Documentation comprises two kinds of information. First of all, it holds documentation to support the reuse of the component. Secondly, it holds information intended for the documentation of the product in which the component will be included. Documentation supporting reuse a) enables the evaluation of each component, b) enables the understanding of the functionality of the component, and c) enables the adaptation of the component for specific needs. The component interface describes the boundaries of the component. The component body describes the internal workings of the component. The test support includes readily available test suites for the component.

The component model also defines relationships between different elements of the model. The realizes-relationship relates analysis, design and the resulting code of a component and this way reflects the possibility of components existing on different levels of abstraction. The includes-relationship relates one or more code components to form a composite object. The RCEvolution-relationship shows components version history by linking different versions of a component.



REUSE MANUAL

1. GENERAL INFORMATION
    - 1.1. Introduction
    - 1.2. Classification
    - 1.3. Functionality
    - 1.4. Platforms
    - 1.5. Reuse status
2. REUSE INFORMATION
    - 2.1. Installation
    - 2.2. Interface descriptions
    - 2.3. Integration and usage
    - 2.4. Adaptation
3. ADMINISTRATIVE INFORMATION
    - 3.1. Procurement and support
    - 3.2. Commercial and legal restrictions
    - 3.3. History and versions
4. EVALUATION INFORMATION
    - 4.1. Specification
    - 4.2. Quality
    - 4.3. Performance and resource requirements
    - 4.4. Alternative components
    - 4.5. Known bugs/problems
    - 4.6. Limitations and restrictions
    - 4.7. Possible enhancements
    - 4.8. Test support
5. OTHER INFORMATION
    - 5.1. System documentation
    - 5.2. References
    - 5.3. Reading aids

FIGURE 6.4 Outline of Sametinger's (1997) Reuse Manual (pp. 206-209).

6.3.3 Sametinger's (1997) Reuse Documentation

In addition to the documentation of software there must be reuse documentation for software components. To effectively and correctly reuse a software component there should be information that enables

- the evaluation of component
- the understanding of the components functionality,
- the use of the component in a certain environment, and
- the adaptation of the component for specific needs.

Regular software documentation does not fulfill these needs. The component is not reusable without proper documentation. Thus, documentation must be valued as an essential part of a software component. Sametinger has elaborated his Reuse Manual on NATO's (1993) standard and Karlsson's book (Karlsson, 1995) (Figure 6.4). Additionally, he has used also Krueger (1992) and Meyer (1994) as the main references.

6.3.4 Shortcomings of the models

Each model basically recognizes the same general purpose for component documentation that supports the reuse process. However, some differences can be found in how the models relate to the overall field of software documentation. The NATO (1992) model emphasizes that, in addition to explicit guidance to the potential reuser, component documentation must fulfil ”the traditional role of documentation” (p. 8-1). Sametinger (1997) clearly distinguishes between other software documentation and his component reuse manual. In Karlsson's (1995) REBOOT model, the reuse-related aspects are partially embedded in the component documentation, and partially documented elsewhere. That is, in addition to documentation explicitly tagged as ”reuse documentation supporting the reuse of the component” (p. 84), Karlsson introduces separate documents for qualifying or administering components (which naturally support the reuse of the component as well). It seems that no clearly enacted demarcation yet exists for the genre system of component documentation.

All three models concentrate mainly on depicting the information content of component documentation. Sametinger and the NATO model provide thorough guidelines about what information should be included, whereas Karlsson's model stays on a more abstract level. Anyhow, differences exist, especially on how documentation is structured and what kinds of information are included under particular topics. Hence, a standardized structure for and a detailed set of document genres to be included in this genre system remain to be enacted. Moreover, these three models shed practically no light on how, by whom, and in what order the parts of component documentation should be created and updated. The sequence of the parts of component documentation thus needs to be better illustrated in order to increase general understanding of this (obviously rather complex) genre system.

Little is said about what is expected from communication media and other issues of the actual form concerning this genre system. The models seem to assume that component documentation mostly consist of digital text. NATO (1992) explicitly emphasizes that the documentation should be in a machine-readable form, independent of any particular word processor. Sametinger (1997) and NATO (1992) both highlight conformance with existing general-level documentation standards, and emphasize the fact that the writing should generally be clear, understandable, and complete. The lack of detailed responses to the how-aspect (especially communication media) seems natural because this aspect concretizes only when actual component repositories with adequate documentation are implemented and studied in real organizational contexts.

In all models, the primary stakeholder is ”the reuser” (NATO, 1992, Karlsson, 1995) or ”the software engineer” (Sametinger, 1997). In addition, Karlsson and the NATO model explicate, respectively, that ”the library staff” and ”the repository managers” also need to use component documentation. However, these models consider the question of who should produce the documentation either selfevident (i.e., the producer of the component is always assumed to produce the documentation as well) or otherwise too trivial to be discussed. Karlsson, however, adds that the reuser should produce comments on previous use of a component to be included in the documentation, thus implying the possibility for various contributing stakeholders. The who-aspect of component documentation remains rather unproblematized. Furthermore, none of the models explicates temporal aspects of component documentation.

With regard to the physical and logical location of the information, all three models denote that a component documentation should constitute a self-contained unit, i.e., it should not be embedded in one big document describing a set of components (Sametinger, 1997) or other surrounding documentation (NATO, 1992). The models also assume that a component documentation should be placed in a logically organized (digital) repository. Interestingly, NATO (1992) points out that the actual (code) component can also be physically located elsewhere, for instance in a separate organization, as long as the documentation includes information about this location.

To summarize, the three models have focused mainly on the information content to be included in component documentation, neglecting the communicative viewpoint to a large extent (which, however, represents the major rationale for the documentation in the first place). The next section attempts to elaborate this viewpoint by the means of genre theory.

6.4 Genre System of Component Documentation

This section first elaborates component documentation model based on Lim's (1997) reuse process. We apply the concept of genre system to extract requirements for communication in the reuse process. A result from this scrutiny resides in the argument that documentation needs two parts: 1) reusable part, and 2) part that enables reuse. To illustrate our abstract model we also give a short example of using it in a real world context.

6.4.1 Elaboration of the model for documenting components

This section elaborates an abstract-level genre system of component documentation. To demarcate the why-aspect of this genre system, we refer to Lim's (1997) typology of the reuse activities: production, brokering, and consuming a component, and managing the reuse process. These activities necessitate that component documentation should not only cover the reusable part of a component. In addition, documentation that supports communication in the whole reuse process is needed. In the following, we first declare the issues related to documenting the reusable part of the component (conducted in the production activity to support the consuming activity). After this we outline the documentation needed for supporting reuse in the activities of brokering, consuming, and managing components. Finally, we summarize our model in light of the framework for analyzing genre systems, and discuss the model with regard to the shortcomings of the previous models.

The reusable part of a software component becomes embedded in future software systems to be developed, i.e. it ends up as a part of the design documentation and/or programming code. It can be used in three separate ways during a component-based development effort (Table 6.2). When a component becomes a part of software under development it also serves as an input for the next phase of development. For example, if the software developer in the analysis phase decided to include an analysis component as a part of the solution, they would seek similar information from an existing component as they would from a component created from scratch. The developers in the design phase could also utilize the analysis component as an input. Finally, the analysis component can be tested in the testing phase as a part of the whole software. In order to use a component as a part of software, the developers must, therefore, comprehend the objective of the component. They also need information about alternative solutions that were considered but rejected and, of course, the rationale for the rejections, i.e., the design rationale for the component (Freeman, 1983).

Besides the reusable part, a component should also include documentation that supports its reuse in the activities of consuming, brokering, and managing the reuse process in the software development organization. Table 6.3 summarizes the tasks, stakeholders, and documentation elements required.

TABLE 6.2 The ways to use component, stakeholders and documentation elements of the reusable part of software component.

Reuse activity	Component use	Stakeholders	Documentation elements
Using component	- As a part of the solution in the development phase at hand, - As an input for the subsequent development phase, and - As an input for testing the software.	- software developer using the component as part of the solution - software developer using the component in the sub-sequent phase of the development process, and - software tester who tests the results from the development process.	- the objective of the reusable component, - the design rationalization to create the component - the reusable component as such, i.e., the results of the production work, and - the test procedures to certify the correctness of the component (used to certify that the component works correctly – not how the component has been tested for the purposes of reuse).

TABLE 6.3 Tasks, stakeholders and documentation elements to support the reuse of a component.

Reuse activity	Tasks in the reuse process	Stakeholders	Documentation elements
Consuming	- find the suitable component(s), - select the most suitable component for use, - adapt the component for use, and - integrate the component into the software solution(Biggerstaff and Richter, 1987; Taivalsaari, 1993; Lim, 1997).	The software developer who wants to utilize a component uses consuming documentation. The producers of the reusable part of the component (or other closely related stakeholders) should initially create this information. However, the consumers of the component may update consuming information when they find new ways to use, adapt, and integrate the component (Karlsson, 1995).	- information to find the suitable component(s), - information to select the most suitable component for use, - information how to adapt the component for use, and - information about integration of the component as a part of the software solutions.

continues

TABLE 6.3 *(continued)*

Reuse activity	Tasks in the reuse process	Stakeholders	Documentation elements
Brokering	- assessing a component, - procuring a component, - certifying a component, - adding a component, and - deleting a component(Lim, 1997; see also STARS, 1993).	The broker can be an individual who is assigned to these tasks. More often, brokering may involve several people and even separate groups of people. For example, one group assesses and procures components whereas another one is assigned to the certifying task. Usually, the maintainer of a repository is responsible for adding components into the repository or deleting components from the repository. In case of deletion or addition there are also other stakeholders who should be informed about the changes; there should thus exist defined procedures for these cases.	- information about assessing the component, - information about procuring the component, - information about certifying the component, - procedures to add the component, and - procedures to delete the component.

continues

TABLE 6.3 *(continued)*

Reuse activity	Tasks in the reuse process	Stakeholders	Documentation elements
Managing the Reuse Process	Managing the reuse process includes tasks that set goals and rules to support reuse (Lim, 1997). Management tasks vary in companies but they should reflect the objectives for the reuse, e.g. establishing reuse metrics (Frakes and Terry, 1996) and following the reuse level (Freeman, 1983). It is also important to monitor the users of a component so that, in case of deleting a component, relevant people could be informed, for example. Looking after the price and security of a component also involve information related to the management of reuse.	The managing activity can involve several stakeholders closely related with the reuse process. The producers of a component may set the security level of a component so that only people inside one department are allowed to use the component. The users of a component can add their name to the list of users. Furthermore, other people may be responsible for pricing the components and monitoring their use.	- information about the creator, - information about the support, - information about the pricing policy, - information about the security level, and - information about the users.

We outline further our model in light of the analytical framework of genre systems (Orlikowski & Yates, 1998), and compare it with our analysis of the previous documentation models.

Why? Component documentation should support communication in and among all activities of the reuse process. This view is more or less equivalent to the previous documentation models (as it naturally should be). However, our model explicates a demarcation of this communicative genre system based on

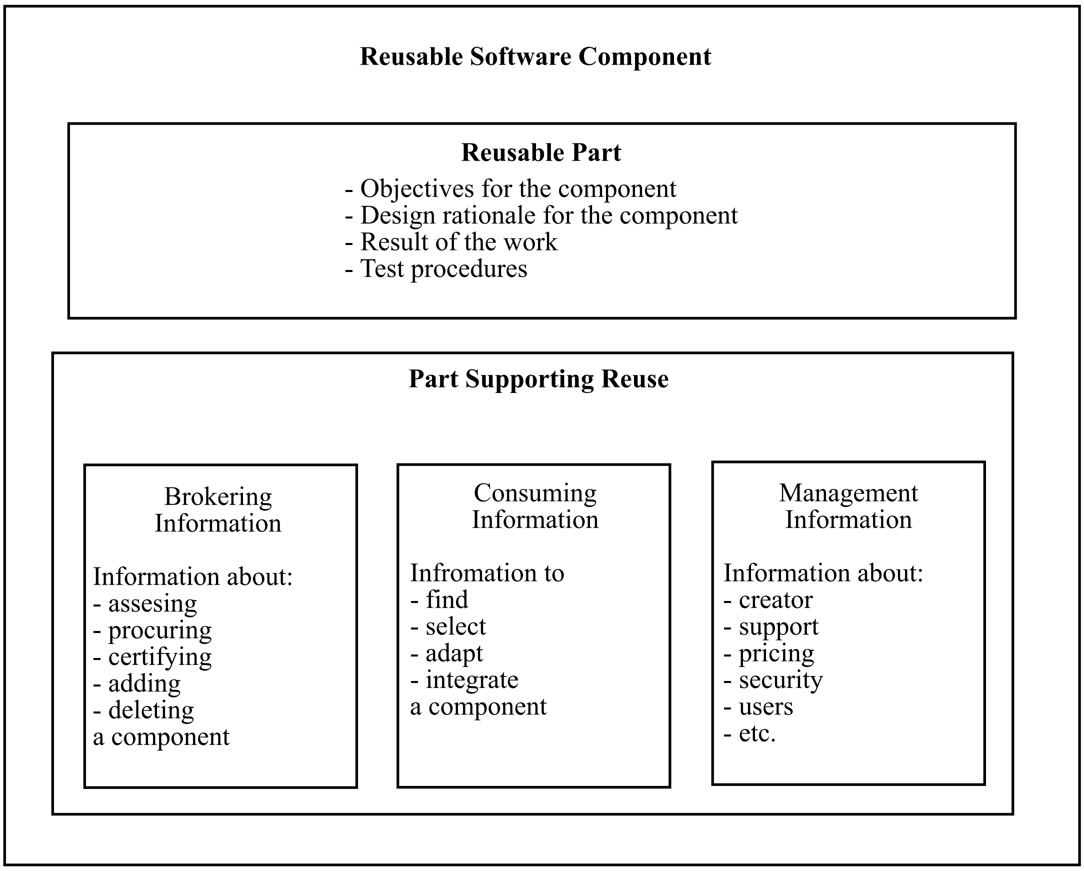


FIGURE 6.5 Information structure for component documentation.

Lim's (1997) concise model of the generic reuse process. This sharpens understanding of the general nature and sphere of this genre system compared to the previously ad hoc structured and diverging declarations of component documentation.

What? Figure 6.5 illustrates a general-level information structure for documenting reusable software components. Our model does not bring in any new information in the sense that the subparts (or constituting genres at a topic level) are already more or less identified and divergently emphasized in the previous models – our model just draws the issues mentioned in each of them together (see Appendix 1). Moreover, unlike any of those, our model explicitly covers information needed in all activities of Lim's (1997) generic reuse process, and the general-level structure of documentation draws on these activities, providing a simple but covering structure to continue scrutiny about their organization-specific implementations. The earlier models followed no specified logical basis for structuring component documentation.

If we consider more closely communicative actions related to the subparts of component documentation, we can observe the following sequence. First, the producer creates the reusable part. After this the producer tries to envision how this component can be made usable for potential consumers. Next, the component is analyzed by the broker. The broker assesses, procures, and certifies the component. If the component is accepted and added to the repository, then relevant stakeholders, such as the manager, are informed. The reuser (consumer) utilizes all parts. Moreover, the role of a reuser is also emphasized when consuming and management information are updated. In some cases, the producer may (re)use consuming information as a basis for developing a new version of the component etc. Even this coarse sketch might illustrate the manifold needs for using component documentation according to the varying stakeholder tasks and roles, which should be considered when these kinds of communicative documentation systems are implemented in practice – a fact that is not sufficiently emphasized in the previous literature. As the roles of actual stakeholders may vary organization-by-organization, the detailed genres included in this genre system could be defined contextually by checking their content against the generic information structure in the figure 6.5. This is illustrated further in the example below.

How? Even our, thus far, abstract model clearly implies a few requirements for implementing this genre system in a software-producing organization. Several stakeholders, potentially scattered across large global organizations, can be involved in producing information for and using it from different subparts of this genre system. The implementation should thus support the clarification of access rights and the authentication of the editors and readers of the documents. The revisions of consuming and management information, and the reusable part itself, require also technological support for managing the consistency of the whole package of this information. Some of the elements in the documentation may be identical among several components, for example, information about pricing policies. In these cases it is natural that this kind of information is kept in one place and a component only refers to this information (cf. Karlsson, 1995). Hence, e.g., hypertext solutions supporting links between a component documentation and more general-level documents in a repository may be needed. Since a large proportion of components is often produced in other organizational units, or even in other companies, intra- and extranet applications are not out of scope here. In conclusion, the ideal technological solution for this field would operationalize the functionality of modern content management systems (cf. Boiko, 2001); including standardized document structures and technologies for processing structured and semi-structured content, high-level information security, effective version and configuration management (combining documentation and software), and effective sharing practices via intra-/extranet solutions. The more detailed aspects of the actual forms and implementations remain to be scrutinized further elsewhere. Still, this discussion has already significantly concretized this problem area compared to the earlier models of component documentation.

Who? The producers of a component (or stakeholders closely related to them, such as specialized documenting staff) create documentation for the reusable part. They also create information for consuming and partly for management. The brokers create brokering information. The reusers create additional consuming information and partly information for the reuse management (in addition to distinguished repository managers).

To whom? Documentation is evidently created primarily for the reusers, but also for the brokers and reuse management. The reusable part is used by the reuse-oriented developer who first utilizes the component, potentially by the developers in the subsequent phase of the software development effort, and by the software testers. A detailed and context-dependent explication of these stakeholder roles is needed to implement component documentation in practice in any target organization – this aspect has been left unproblematized in the previous literature on the topic.

When? No component exists without component documentation. Hence the first version of the documentation has to be made in immediate connection with the reusable part. After this, a component documentation should be flexible enough to evolve continuously along with actual reuse occasions and revisions of the reusable part. Actual implementations of this genre system might reveal more detailed and context-dependent temporal rules and expectations, for example, deadlines for assessing the component and expectations connected to revisions. This aspect must be kept in mind when implementing instances of this genre system. Since the previous models explicated no temporal aspects, even this abstract-level recommendation contributes to the field.

Where? Not much can be said at an abstract level about the spatial or logical locations to organize a component documentation – except the fact already stated in the previous models that they should be stored in a (digital) repository. This dimension must be considered in more detail when implementing actual instances of this genre system. Organizational boundaries and geographical distribution of a particular organization especially affect the actual physical storage place of this documentation. Still, the logical location in the ”cyberspace” will emerge as the most important question.

6.4.2 An example of the model use

The component documentation model was applied within a project at TietoEnator Ltd, Finland, the leading supplier of value-added IT services in Europe. The project in question applied Hierarchical Domain Analysis (HDA) (Forsell 2001a, 2001b) that aims at improving the creation and use of reusable components. One crucial part in HDA is to document the components properly to make them reusable. In 2001, the project had documented four components according to the model. Three of them, ’Job Queue’, ’Rule-based Processing of a Transaction’, and ’Sales Ledger’ are at the design level. The fourth, named ’Contract’ is currently released as a stateless Enterprise Java Bean. The model was used for discussing whether all necessary information elements of component documentation had been taken into account. First we describe the reuse process involved with creating the component and accompanying documentation. After this we show, as an example, how the specific genre of ”design pattern documentation” was emerging as a part of the component documentation.

The reuse process starts with a domain analysis (DA), in which a group of domain analyzers looks for appropriate components and creates a domain model. The group first searches for candidate components, to be reviewed by a steering committee. The steering committee decides on which candidate components should be refined further. The approved candidates become the actual reusable components, being documented as design patterns by the DA group. A component is considered ready after the DA group's evaluation and approval. It is also possible to use a third party evaluation. The resulting reusable software component is placed into the domain model as well as to the component repository (this represents the brokering task of the component). In the case of consuming task, the reusers browse the domain model or conduct key word searches to the repository. The reusers decide, based on the documentation, whether the component fulfils their needs. If the component is regarded as usable, the component documentation aids reusers in adapting it to the software.

The roles of stakeholders are identified based on the reuse process. The reuse process defined by the target organization states that there are people creating components (DA group), brokering it (steering committee, repository representative), and consuming it (reuser, and implicitly the customer). Most probably, these roles vary in different organizations, according to a number of options to implement the reuse process in practice.

We identified five genres of component documentation in connection to the target organization's (rather fresh) reuse process thus far: 'Domain model', 'Design pattern documentation', 'Steering committee approval', 'Review', and 'Expert evaluation'. In addition to those, the repository was used also to retrieve information for some tasks dynamically during the reuse process: i.e., we could not yet identify explicitly structured genres in all tasks, although the repository was to be used in those tasks. This implies that the reuse process as such is only on its way toward a fully defined one in the target organization (another question is whether it never could or should reach such fully defined status). Table 6.4 summarizes how the tasks of the reuse process relate to the genres of (and the ad hoc information needs for) component documentation identified in the target organization.

As a more detailed example, let us discuss one genre of the overall documentation, that is, the design pattern documentation for a component (Appendix 2). A design pattern document is typically written with MS Word and the structure follows Gamma's et al. (1995) idea of design pattern documentation. A design pattern is a named solution to a recurring problem in a particular context of object-oriented design (Vlissides, 1998). The basic design pattern form of Gamma et al. (1995) was, however, slightly altered here. The "Where to use" section was based on the DA, describing for the possible reusers where the component was seen usable within that domain by the DA group. Furthermore, the "Example use" section is based on the domain analysis as well.

The reusable part, which becomes a part of the resulting system, is presented in the section "Structure". The other sections comprise the part that enables the use of the component. Information for component retrieval can be

TABLE 6.4 Tasks of reuse process and corresponding genres in the example case.

TASK of the reuse process	GENRE (Repository(*) means that no specific genre was identifiable for that task, but documented content was retrieved dynamically from the repository in an ad hoc manner)
Creating component	
- analyzing domain	Domain model
- producing component	Design pattern documentation
- Maintaining and enhancing component	Repository(*)/Design pattern documentation
Brokering component	
- Assessing components for brokering	Steering Committee Approval
- Procuring components	Steering Committee Approval
- Certifying components	Reviews, Expert evaluation
- Adding components	Repository(*)
- Deleting components	Repository(*)
Consuming component	
-identifying System and component requirements	N/A
- locating components	Domain model/Repository(*)/Design pattern documentation
- assessing components for consumption	Design pattern documentation
- understanding components	Design pattern documentation
- integrating components	Design pattern documentation

TABLE 6.5 A summary of the genre system for component documentation model.

Why?	Documentation is created to support the reuse process so that all relevant documentation for supporting reuse would be supplemented.
What?	Documentation has two parts: the reusable part and part supporting reuse. The reusable part includes the component that comes as part of the new software. The part supporting reuse includes information about managing, brokering, and using the component. Five genres could be identified ('Domain model', 'Design pattern documentation', 'Steering committee approval', 'Review', and 'Expert evaluation'), which contribute to the different tasks of the reuse process. Moreover, the repository is used in ad hoc ways in some tasks, for which no enacted genres could be identified at the moment
How?	The documentation about the component includes information about the reusable part and part supporting the use of the component. The related documents are attached automatically by the repository to the information about managing and brokering the component.
Who/m?	The component documentation is created for the reusers, reuse managers and people responsible for brokering the component. Each of these groups has specific information needs of its own.
When?	Component reuse takes place in connection to software development projects.
Where?	Component documentation is kept in a repository.

found in the section "Where to use". The sections "Purpose" and "Motivation" support the selection task. The reusable part, i.e., the structure, can also help in the selection. The adaptation and integration is guided with the section "Example use".

Table 6.5 summarizes the genre system of component documentation in the target organization.

6.5 Implications

In summary, our model implies that at least the following dimensions should be considered in connection with inquiring and implementing the genre system of component documentation in practice:

1. Process: our model denotes that a component documentation is not a static entity, but rather a communicative process of several stakeholders, continuously evolving with time alongside the uses and revisions of the com-

ponent. Instead of ad hoc structuring of this genre system, the explicated activities of the reuse process (Lim, 1997) offer a solid conceptual ground to structure and implement documentation systems for software components. In our example we illustrated the document model in a real world context, demonstrating how certain genres could already be identified and recognized in connection to certain tasks. This helps structure the work, communication, and related information production and retrieval in the reuse process and CBD in general.

2. Stakeholder roles: our model offers a motivation and basis for explicating stakeholder roles, with their rights and responsibilities, related to component documentation at a detailed level before implementing a system of this kind in the organizational context in question. The explication of processes and roles together constitutes a challenge to organizational design related to the component documentation system before its practical implementation. The example illustrates that component documentation affects a number of stakeholders: the creators, reusers, and brokers of a component, and a steering group to accept the components to be included in a repository. Moreover, information about the costs and uses of a component are needed at the management level.

3. Detailed structuring and standardization of the information content of component documentation. As the production of software components (especially code components) increasingly takes place outside the administrative borders of those organizational units potentially consuming the components, the need to standardize the contents of component documentation at the corporate level, or even at the industry-level, seems obvious. This standardization process would not necessarily emerge as an easy one even within one organization, let alone within an organization network of CBD. Our difficulties to identify genres to utilize the repository in connection to the tasks of maintaining, adding, deleting and locating components in the target organization might illustrate this issue. The component documentation model that is explicitly based on the reuse process model could, however, offer a solid conceptual basis for negotiating this standardization in and among software development organizations further, without forgetting the viewpoint of different stakeholders and their needs for communication. More experience on the repository-level vis-à-vis component-level documentation would be needed for crystallizing this aspect in the future.

4. Several technological challenges seem to relate particularly to the utilization of content management functionality, such as version and communication management, automatic publishing, access rights management and information security, link management, and all the other challenges of managing heterogeneous information content in the digital era (e.g., Boiko, 2001) with regard to component documentation. In the target organization, a repository was definitely considered useful and necessary for stor-

ing and restoring the components. A challenging issue, however, would be its management along time in varying reuse processes and for varying kinds of more or less accepted genres of component documentation among the varying assemblies of stakeholders.

6.6 Conclusion

Our genre system based analysis of previous literature revealed that several aspects of component documentation have remained implicit in the literature, and thus need further elaboration. We have elaborated a genre-system based model for component documentation, drawing explicitly on the generic model of the reuse process (Lim, 1997), as it supports the reuse of all kinds of software elements (requirements, design models, code). The reuse-process-based documentation model supports the reuse of white-box components as well as black-box components. The model can be applied as a basis for designing component documentation solutions for CBD in software-producing organizations. Specifically, our model implies that the following aspects should be explicitly scrutinized and contextually tailored: the reuse process, stakeholder roles, standardized contents and structures for the subdocuments included, and several technological challenges related to this complex genre system. Special characteristics of documenting different kinds of components that result from the different phases of software development should be investigated further – as well as relationships between component-level and repository-level genres aimed at supporting reuse. Also, the versioning of a component documentation as a holistic genre system, with varying potential genres and their relationships, should be studied further. We regard our model as a theoretically grounded foundation for further research and elaboration; for instance, for proactive action research initiatives aimed at establishing better documentation practices in actual software-producing organizations, or for industry-wide initiatives to define documentation standards for component-based software development.

References

Arango, G. (1989), "Domain Analysis – From Art Form to Engineering Discipline," Proceedings of the 5th International Workshop on Software Specifications and Designs, 1989, pp. 152-159.

Auramäki, E., Lehtinen, E., Lyytinen, K. (1988), "A speech-act-based office modeling approach," ACM Transactions on Office Information Systems, Vol. 6, No. 2, 126-152.

Auramäki, E., Lyytinen, K. (1996), "On the Success of Speech Acts and Negotiating Commitments," in Dignum, F., Dietz, J., Verharen, E., Weigand, H. (eds.), Communication Modeling – The Language/Action Perspective:

Proceedings of the First International Workshop on Communication Modeling (LAP ’96), London: Springer.

Austin, J.L. (1962), How to do things with words. Clarendon Press, London.

Bakhtin, M., (1952/53), “The Problem of Speech Genres,” in Emerson, C., Holmquist, M. (eds.) (1986), Speech Genres and Other Late Essays, English transl. from Russian by McGee, V.W., Austin: University of Texas Press, 60-102.

Basili, V., Abd-el-Hafiz K. (1996),“A Method for Documenting Code Components,” Journal of Systems Software, Vol. 34, 89-104.

Bazerman, C. (1994), “Systems of Genres and the Enactment of Social Intentions,” in Freedman A., Medway, P. (Eds.), Genre and the New Rhetoric London: Taylor & Francis, 79-101.

Biggerstaff, T., Richter C. (1987), “Reusability Framework, Assessment, and Directions,” IEEE Software, March, Vol. 4, No. 2, 41-49.

Boiko, B. (2001). Content Management Bible, New York: Hungry Minds.

Brown, J.S., Duguid, P. (1994),“Borderline Issues: Social and Material Aspects of Design,” Human-Computer Interaction, Vol. 9, No. 1, 3-36.

Conger, S., Schultze, U. (1999), “Understanding e-Commerce through Genre Theory: The Case of the Car-Buying Process,” in Ngwenyama O., Introna L.D., Myers M.D., DeGross J.I. (Eds.), New Information Technologies in Organizational Processes: Field Studies and Theoretical Reflections on the Future of Work, Boston, Kluwer, 219-239.

Daft, R.L., Lengel, R.H. (1986), “Organizational Information Requirements, Media Richness and Structural Design,” Management Science, Vol. 32, No. 5, 554-571.

Dusink, L., van Katwijk, J. (1995), “Reuse Dimensions,” Software Engineering Notes, August 1995, Proceedings of the Symposium on Software Reusability, Seattle, Washington, April 28-30 1995, 137-149.

Forsell, M. (2001a), “Using Hierarchies to Adapt Domain Analysis in Software Development,” in Sein, M.K., Munkvold, B.E., Ørvik, T.U., Wojtkowski, W., Wojtkowski, W.G., Zupančič, J. (eds.), Contemporary Trends in Systems Development, Papers presented at ISD2000, the Ninth International Conference on Infomation Systems Development: Methods and Tools, Theory and Practice, held August 14-16, 2000 in Kristiansand, Norway,Kluwer Academic/Plenum Publishers, New York, 105-118.

Forsell, M. (2001b), “Adding Domain Analysis to Software Development Method,” Accepted for presentation at Tenth International Conference on Information Systems Development, ISD2001, London, United Kingdom, 5-7 September 2001. To be published by Kluwer Academic/Plenum Publishers 2001.

Frakes, W., Pole, T. (1994), “An Empirical Study of Representation Methods for Reusable Software Components,” IEEE Transactions on Software Engineering, August, Vol. 20, No. 8, 617-630.

Frakes, W., Terry, C. (1996),“Software Reuse: Metrics and Models,” ACM Computing Surveys, June, Vol. 28, No. 2, 415-435.

Freeman, P. (1983), “Reusable Software Engineering: Concepts and Research Directions,” ITT Proceedings of the Workshop on Reusability, 129-137.

Fulk, J., Schmitz, J., Steinfield, C.W. (1990),“A Social Influence Model of Technology Use,” In Fulk, J., Steinfield, C. (eds.) Organizations and Communication Technology, Newbury Park: Sage, 117-140.

Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1995), Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Publishing Company.

Gordon, M.D., Moore, S.A. (1999), “Depicting the Use and Purpose of Documents to Improve Information Retrieval,” Information Systems Research, Vol. 10, No. 1, 23-37.

Henninger, S. (1997), “An Evolutionary Approach to Constructing Effective Software Reuse Repositories,” ACM Transactions on Software Engineering and Methodology, April, Vol. 6, No. 2, 111-140.

Karlsson, E. (1995), Software Reuse: A Holistic Approach, Chichester: John Wiley and Sons.

Krueger C. (1992), “Software Reuse,” ACM Computing Surveys, June, Vol. 24, No. 2, 131-183.

Lam, W., McDermid, J. (1997) “A Summary of Domain Analysis Experience by Way of Heuristics,” Proceedings of the 1997 Symposium on Software Reusability, 1997, pp. 54-64.

Lim, W. (1998), Managing Software Reuse, Upper Saddle River: Prentice Hall.

Markus, M.L. (1994), “Electronic Mail as the Medium of Managerial Choice,” Organization Science, Vol. 5, No. 4, 502-527

McIlroy, D. (1968), "Mass Produced Software Components," Report on a conference by the NATO Science Committee, Garmish, Germany, October 7-11 1968, in Naur, P., Randel, B., Buxton, J. (eds.) (1976), Software Engineering: Concepts and Techniques, New York: Petrocelli/Charter, 88-98.

Meyer, B. (1994), Reusable Software: The Base Object-Oriented Libraries. Prentice Hall Object-Oriented Series.

Miller, C.R. (1984), "Genre as Social Action," Quarterly Journal of Speech, 70: 151-167, reprinted in Freedman, A., Medway, P. (Eds.) (1994), Genre and the New Rhetoric, London: Taylor & Francis, 23-42.

Mintzberg, H., (1983), Structure in Fives: Designing Effective Organizations. Prentice Hall, Inc., Englewood Cliffs, N.J.

NATO (1992), NATO Standard for the Development of Reusable Software Components, Volume 1 (of 3 Documents), (http://www.asset.com/WSRD/abstracts/archived/ABSTRACT_528.html), (accessed 1 June 2000).

Orlikowski, W.J., Yates, J. (1994), "Genre repertoire: The structuring of Communicative Practices in Organizations," Administrative Science Quarterly, Vol. 39, No. 4, 541-574.

Orlikowski, W.J., Yates, J. (1998). Genre Systems: Structuring Interaction through Communicative Norms. Sloan School of Management Working Paper #4030, MIT, http://ccs.mit.edu/papers/CCSWP205), (accessed 12 July 1999).

Prieto-Díaz, R. (1994), "Historical Overview" in (eds.) Shäfer, W., Prieto-Díaz, R., Matsumoto, M., Software Re-usability, Ellis Horwood Limited, pp. 1-16.

Prieto-Díaz, R., Freeman, P. (1987), "Classifying Software for Reusability," IEEE Software, January, 6-16.

Päivärinta, T. (2001), "The Concept of Genre within the Critical Approach to Information Systems Development," Information & Organization, Vol. 11, No. 3, 207-234.

Sametinger, J. (1997). Software Engineering with Reusable Components. Berlin: Springer.

Shannon, C.E., Weaver, W. (1949), The mathematical theory of communication, Urbana: University of Illinois Press.

Sillince, J.A.A. (1997), "A Media-Attributes and Design-Choices Theory of the Information Technology-Organization Relation", Journal of Organizational Computing and Electronic Commerce, Vol. 7, No. 4, 279-303.

Spinuzzi, C., Zachry, M. (2000), "Genre Ecologies: An Open-System Approach to Understanding and Con structing Documentation," ACM Journal of Computer Documentation, Vol. 24, No. 3, 169-181.

STARS (1993), STARS Conceptual Framework for Reuse Processes (CFRP), Volume I: Definition, Version 3.0, STARS-VC-A018/001/00, Informal Technical Report. (http://www.asset.com/WSRD/ASSET/A/495//ASSET_A_495.tar.gz), (accessed 1 June 2000).

Taivalsaari, A. (1993), A Critical View of Inheritance and Reusability in Object-oriented Programming, Ph.D. Thesis, Jyväskylä Studies in Computer Science, Economics and Statistics 23, Jyväskylä: University of Jyväskylä.

Vlissides, J. (1998), Pattern Hatching: Design Patterns Applied. AddisonWesley Longman, Inc., Reading, Massachusetts.

Yates, J., Orlikowski, W.J. (1992), "Genres of Organizational Communication: A Structurational Approach to Studying Communication and Media," Academy of Management Review, Vol. 17, No. 2, 299-326.

Yates, J., Orlikowski, W.J., Okamura, K. (1999), "Explicit and Implicit Structuring of Genres in Electronic Communication: Reinforcement and Change of Social Interaction," Organization Science, Vol. 10, No. 1, 83-103.

Yates, J., Orlikowski, W.J., Rennecker, J. (1997), "Collaborative Genres for Collaboration: Genre Systems in Digital Media," Proc. of the 30th Annual Hawaii International Conference on System Sciences: Digital Documents, Los Alamitos CA: IEEE Computer Society Press, Vol. VI, 50-59.

Zimmerman, E.N. (1994). "On Definition and Rhetorical Genre," in Freedman, A. and Medway, P. (Eds.), Genre and the New Rhetoric, London: Taylor & Francis, 125-132.

Zmud, R.W., Lind, M.R., Young, F.W. (1990), "An Attribute Space for Organizational Communication Channels," Information Systems Research, Vol. 1, No. 4, 440-457.

Appendix 1

TABLE 6.6 Comparison of three models for component documentation (to support reuse).

	NATO (1992, Section 8): "Documentation of reusable software"	Karlsson (1995, pp. 81-87): "The REBOOT component model"	Sametinger (1997, pp. 203-210): "Reuse documentation / Reuse manual"
Why?	"... serves a dual role; it fills the traditional role of documentation, and also provides explicit guidance to the reuser."	"...information about the component to make reuse feasible." "...describes the information needed for a packaged reusable component."	"... to effectively and correctly reuse a software component"

continues

TABLE 6.6 *(continued)*

	NATO (1992, Section 8): "Documentation of reusable software"	Karlsson (1995, pp. 81-87): "The REBOOT component model"	Sametinger (1997, pp. 203-210): "Reuse documentation / Reuse manual"
What? = the content (the sequence not explicated in any of these models)	- "Normal project documentation" useful to the reuser - Documentation for the Reuse Library (to help the library support classification, identification, and retrieval of the component) (An abstract describing the function, a list of dependencies, classification information, available reusability and quality metrics, outstanding problem reports, recommended enhancements, commercial or legal restrictions, access information) - The Reuser's manual (Introduction: purpose, overview; Function: operation, scope; Interfaces: RSC specification, external references and parameters, interfaces by class; Performance: assumptions, resource requirements, exceptions in responding incorrect inputs, test results, known limitations; Installation: instantiation, interfaces, partial reuse provisions, modification provisions, diagnostic procedures, usage examples; Procurement and support: source, ownership, maintenance; References, Appendices)	Relationships (e.g. a collection of links) to the following information relevant for a particular component - Classification information (to aid identification and retrieval) - Component qualification information (describes the quality and reusability of the component to decide whether a candidate fulfils quality and reusability requirements, also the reuse history of the component) - Component administrative information (general: attributes of the developer, time when developed, inserted, or motivated etc.; authorization: reuse rights, payment methods; pricing) - Documentation (to support the reuse of the component: evaluation in a set of candidates, understanding the functionality, the adaptation for specific needs; a part intended for the documentation of the product in which the component will be included) - Component interface and component body - Test support	An outline consists of: - General information (introduction, classification, functionality, platforms, reuse status) - Reuse information (installation, interface descriptions, integration and usage, adaptation) - Administrative information (procurement and support, commercial and legal restrictions, history and versions) - Evaluation information (specification, quality, performance and resource requirements, alternative components, known bugs/problems, limitations and restrictions, possible enhancements, test support, interdependencies) - Other information (system documentation about the implementation of the component, references, reading aids etc.)

continues

TABLE 6.6 *(continued)*

	NATO (1992, Section 8): "Documentation of reusable software"	Karlsson (1995, pp. 81-87): "The REBOOT component model"	Sametinger (1997, pp. 203-210): "Reuse documentation / Reuse manual"
How?	Machine-readable form. Not dependent on a particular word processor. Completeness, clarity, and understandability particularly important	Not explicated	"compliance with accepted documentation standards; use of consistent structures, styles, and formats; consistency with the code; writing in clear and understandable form; etc."
Who/m?	Who: not explicated Whom: the library staff , the reuser	Who: the reuser may document experiences, the producer not explicated Whom: The reuser, the repository manager	Who: not explicated Whom: "The software engineers who decide whether a certain component fits their needs"
When?	No temporal aspects discussed.	No temporal aspects discussed.	No temporal aspects discussed.
Where?	Documentation for each component should be self-contained. Possibly located in the Reuse Library (the reusable part may as well be located somewhere else).	Self-contained package for reuse. Located in a repository (under a conceptual classification).	Each component has its self-contained documentation – not in big documents describing a set of components! Located in a repository.

Appendix 2

An example of the "Design Pattern Documentation" genre

Intent

Contract design pattern describes common concepts and services that must be used while providing solution to deal with a contract. Also, pattern provides common framework to solve this problem, and gives examples how the solution may be adapted for a specific use.

Motivation

A contract may be for example a contract of sale, a contract of purchase, or a contract for billing. While a contract is under preparation it is assumed that different parties and possible products (i.e., the subjects of a contract) are predefined and they can be referenced.

This design pattern separates the contract's form (e.g., electronic document or paper) and its basic information and functionality that are independent of it. This enables flexibility to add for example electronic signature. Inheriting from core-implementation may extend design pattern. Inheriting the interface the basic implementation may be broaden.

Where to use

Contract is usable in domains that need management of contracts, e.g. labour contract, contract of payment, contract of assignment, and issuing different kinds of permits.

Structure

Figure 1. The Contract design pattern and example how to use inheritance to adapt it in a specific need.

Classes

In this part all of the classes are described more closely. Due to the limited space this part is omitted.

Class Name

Here is the name of the class.

Attributes

Here all of the attributes of the class are described more closely.

Operations

Here all of the operations of the class are described more closely.

Conclusions

Contract design patterns helps understanding the basic concepts and functionality when dealing with different kinds of contracts. Design pattern also defines adaptation and possibilities to broaden either the interface or functionality.

Example

The owner of a vehicle is subject to taxation, e.g., he or she pays the use tax for a vehicle that she or he possesses. If needed the subject to taxation may be transferred to another party. Anyhow, he contract must include this information so that the tax payment is sent to its rightful payer.

To the problem described above, the Contract design pattern is applicable (see Figure 1). The needed roles for Transferor and Receiver, and the contract type Transfer of Subject to Taxation are added with inheritance. There is no need to make any further alternations to the Contract design pattern.

Used design patterns

Core-Representation: Payer and BasicContract are adapted from this design pattern.

References

David C. Hay, Data Models. Dorset House Publishing, 1996, Contracts pp. 95-116 Martin Fowler, Analysis Patterns. Addison Wesley Longman, 1997, Contract pp. 175-180 Hans-Erik Eriksson, Magnus Penker, Business Modeling with UML. Wiley, 2000, Contract pp. 215-218

7 ADDING DOMAIN ANALYSIS TO SOFTWARE DEVELOPMENT METHOD

Forsell, M.[†],“Adding Domain Analysis to Software Development Method”. Proceedings of the Tenth International Conference on Information Systems Development, ISD2001, Royal Holloway University of London, 4-6 September, 2001. (To be published by Kluwer Press in 2002)

©Kluwer Academic/Plenum Publishers, 2002. Reprinted with permission.

[†]Chydenius Institute, University of Jyväskylä, Finland

Abstract

The researchers in the field of software development regard the reuse of components as one possible approach when creating quality software in less time and with fewer people. When components are used and created in the software development, one critical success factor is the use of domain analysis (DA). We report an action case study where the DA technique is first integrated into an existing software development method and then refined based on the experience of using it in a pilot project. The results indicate that our approach produces reusable components across a company-wide domain and eases the use of them in other development projects within domain.

7.1 Introduction

The studies to improve the pace, quality, and cost effectiveness of software development continuously introduce new theories, approaches, tools, and frameworks, among other things. One possible approach is the reuse of software. To put it simply, in software reuse ‘the same thing’ is used more than once [4]. This ‘same thing’ can vary depending on the type and the level of the reuse a company practices. Reuse can be divided into generative and composite techniques

[5]. In composition techniques the developer composes software from atomic parts, i.e., from components. In generation techniques the software is generated from higher-level descriptions or specifications that are produced by a developer. In this study we concentrate on component-based reuse.

Reuse community has introduced one specific approach, domain analysis, to foster reuse in software development [21] (see also [2, 3, 23]). In domain analysis (DA), a group of problems are analyzed in order to find reusable components to solve similar problems in a certain domain. Approaches to domain analysis are not only used to foster generative or component-based reuse but also, for example, to understand a problem domain and to learn about it [23]. A domain may be understood from two quite distinct points of view: as a group of programs or as a business domain. Depending on the point of view, similar problems may be identified by studying existing programs and their similarities, or by examining businesses. [23.]

DA is seen as a prerequisite in successful reuse not only by the researchers in the reuse community but also by the methodologists who have introduced component-based development methods (see e.g., [6, 12, 13]). In these methods, domain analysis comes as one of the first steps in the development process. However, there are two problems. First of all, component-based methods present DA superficially and DA is used only to identify components. Successful reuse implies that software is built from components; it is not enough only to create them. Secondly, textbook approaches are not used as presented and most of the software companies practice local method development [8]. If we integrate DA into existing textbook approach it is most likely useless to software organizations. Ideally we should add DA to company's existing development methods.

Hierarchical Domain Analysis (HDA) technique addresses the problems mentioned above [9]. A short summary of HDA is given in Section 2.3. But HDA is only a conceptual solution and if we are to believe research methods proposed for information systems and software engineering, we should also incorporate new ideas into practice and validate them (see e.g., [16, 18, 22]). Thus, we want to apply HDA in industrial setting. The research question is: ”How HDA works in practice and should it be enhanced?” The question implies that first we integrate HDA into an existing development method and then find out its practicality in a software development project. Possible flaws in thought and resulting enhancements must be documented. In this study we first integrate HDA into an existing development method, namely Tieto Object. After this we use HDA technique in a pilot project.

The structure of this study is as follows. Section 2 focuses on the research methods and the study environment, and gives an overview of HDA as well. Section 3 starts with the introduction of the Tieto Object method. After this, we present the results. Because the study is conducted in two parts, it produces two types of results: Local method development and the use of HDA in a pilot project. We also reflect on our experiences about both of these results at the end of Section 3. Finally, the conclusions are presented in Section 4.

7.2 Research Method and Environment of the Study

In this Section we first describe our approach to the research question. After this we describe the organization of the research and the research environment. Finally, we summarize the HDA technique.

7.2.1 Research Method

The objective of this study is to give evidence of the practicality of HDA. Action research gains understanding about some phenomena and it can be used to prove new theories and refine them in practice. We use the action case approach [22] applying it to HDA in industrial setting. Vidgen and Braa argue that action case approach differs from action research in the following ways. First of all, duration of the study is shorter in the former. Most often an action case study takes only a few months whereas a full fledged action research may take several years to conduct. In addition, action case usually concerns a small-scale intervention that is focused and deliberate. Action case approach produces practical knowledge and at the same time gains understanding of the context of the change. In contrast to case research the most dominant feature is that the researcher is an active participant in the research process.

The study was conducted in two parts. In the first part, local method development, DA method was integrated into TietoEnator's in-house development method Tieto Object. This took place from August 1999 to May 2000. The second part, a pilot project, uses HDA technique. This took place from September 2000 to December of the same year. In the first part we used local method development framework as proposed by Tolvanen [20]. Action research approach suits well for this purpose [20]. In the pilot project we followed the technique defined. We collected experience in line with Leavitt's [14] diamond model. Leavitt's model suits to identify changes in organization.

7.2.2 Organization and Environment of the Study

This study is a part of the PISKO project that aims to improve component-based reuse in participating software organizations. The project started in January 1999 and it will end in December 2001. Tekes Technology Development Centre and four participating companies fund the project. This study takes place in one of the participating organizations, TietoEnator Corporation . With a staff of 10,000 and an annual turnover of 1.2 billion euros, TietoEnator is a leading supplier of value-added IT services in Europe and it is the largest software company in Scandinavia. This pilot project is integrated within one of the TietoEnator's own. The customer of this project is the Finnish Administrative Centre of Vehicles, AKE (Finnish acronym).

AKE is a nation wide organization, that handles large amount of data related to vehicles and drivers licenses in Finland. AKE has subcontracted TietoEnator to modernize its current programs to take advantage of new WWW and mobile technologies. New systems have to be adaptable for the challenges of this millennium. Within five years all data systems in the AKEs environment are supposed to be modernized.

TietoEnator has an in-house development method called Tieto Object. TietoEnator has done a substantial amount of work in the area of reuse. Processes needed have been created and software acquired to foster reuse in the organization. Now the point have been reached where domain analysis could bring some advantage in finding more relevant components, and where hierarchical domain analysis could aid in using these components.

This study has been conducted by the author together with an expert from TietoEnator. The expert participates in the work of a TietoEnator's task force to enhances and further develop Tieto Object.

The research environment and objectives are optimal to try out HDA approach for number of reasons. First of all, there exists a system to teke care of the current data manipulation. Secondly, new system will be built on integrated middleware. Thirdly, all current systems will be changed within the next five years so we can collect data about HDA from quite a long period. In this study, however, we present the results from the first pilot project.

7.2.3 Summary of Hierarchical Domain Analysis

Hierarchical domain analysis (HDA) is a domain analysis technique that helps company-wide reuse [9]. It is planned to be useful not only in defining components but also in the process of finding and using these components during development. Thus HDA should be part of every development project that wants to take an advantage of the component reuse. HDA is based on the Common Process of Domain Analysis as proposed by Arango [3]. In HDA, a domain is seen company-wide. This means that target organization as a whole is a basis for the domain model. HDA has five phases, and these are further divided into separate steps (Table 7.1).

The first phase is done only once in each organization. Here a company-wide domain model is created. In the domain model we have the business aspect and the software aspect. The business aspect reflects the company's structure and its way of doing business. The structure of the company forms a hierarchy of sub-domains. A sub-domain is part of some larger domain, and sub-domain identification is based on company's structure. A company can group its units in two ways: by function performed or by market served [17]. With this, HDA wants to point out that software should serve company's objectives, and that software supports and adds value to the company. The software aspect reflects the structure the software-solutions must be based on. Often the business software structure consists of middleware and system software.

The second phase focuses on the current development effort and identifies the sub-domain it belongs to. Because we have already modeled the company-wide domain, determining current project and its sub-domain should be easier. This phase is, however, argued to be one of the hardest part of DA (e.g., [19]).

TABLE 7.1 Hierarchical domain analysis' phases and steps.

1. Create domain model for the company
1.1 Identify company's key processes
1.2 Determine the rationale for software development
1.3 Create the hierarchy of sub-domains for the business aspect
1.4 Create the hierarchy of sub-domains for the software aspect

2. Define sub-domain
2.1 Define the boundaries of the current development project
2.2 Find information sources for the sub-domain
2.3 Specify the problems and requirements in the sub-domain

3. Analyze business aspect
3.1 Define the business processes
3.2 Define the key concepts
3.3 Define the needed services
3.4 Analyze the identified services
3.5 Classify the services

4. Analyze software aspect
4.1 Define the layers for the software aspect
4.2 Define the used software solutions
4.3 Classify the found solutions

5. Use domain analysis results

The third phase identifies components based on the services that software must offer in defined sub-domain in order to support software users' and business' needs. Also, components are classified so that any subsequent projects can find and use them. The objectives of this phase are twofold. First, reusable components are identified and defined. Second, possible reuse opportunities are identified from the domain hierarchy. The hierarchy is traversed backwards with each identified component to find out if an opportunity for reuse exists in other sub-domains. We raise the component in the hierarchy to the level where we can see reuse opportunities.

The fourth phase explores possible solutions more carefully. Software solutions, that are used to create and are bases for new software system, are identified. The reason for this phase is that software solutions used limit the possible communication between components, i.e., components can not communicate by any other means than what middleware or system software allow. Pre-existing middleware and system software solutions must be used when we want to create code components and interaction between them. The fifth phase is not exactly a phase but it is presented here because we want to emphasize the importance of separating the creation and the use of components. The results can be used in the subsequent phases of the software development process.

7.3 Results

The results of the method integration part and pilot use part are reported separately in their own sub-sections. Sub-section 3.1 presents the way HDA technique was integrated into Tieto Object. We start by summarizing Tieto Object. Then we describe the steps we took during the integration of HDA into Tieto Object. Sub-section 3.2 describes our experience of using HDA in the pilot project. We explain how we conducted each phase of HDA and what was altered in the original approach. In sub-section 3.3 we point out some relevant experiences about the method integration and the pilot project, which we think might be useful when implementing similar projects in other organizations.

7.3.1 Integration of Hierarchical Domain Analysis into Tieto Object

Tieto Object is an object-oriented method and it covers a large portion of systems development phases:

1. Business Process Reengineering/Requirements Engineering
2. Analysis
3. Design
4. Implementation

The phases of Tieto Object method require some explanation. First of all, testing is an essential part of every phase. The results are formally checked before they can be used in a subsequent phase. Second, maintenance is an operational mode in itself and it does not belong to Tieto Object method. Finally, first phase may be reduced to include requirements engineering only. Requirements engineering may be part of the analysis phase depending on the expected difficulties in the task. Further, a project may only be an analysis or a design project and subsequent phases may be omitted.

Tieto Object is a toolbox method in contrast with cookbook methods. This means that Tieto Object does not rigidly prescribe the steps and in which order they must be taken. Tieto Object gives a number of techniques that may be used during systems development and shows how these relate to each other. The toolbox approach eases the task of integrating HDA into Tieto Object. We have to find the place for HDA in the overall process and define its inputs and outputs.

The first task when deploying any domain analysis method is to define why, in the first place, we are trying to achieve the reuse of the components. Second, we must define what kind of components we want to use. These two considerations influence later on to the integration of the domain analysis method into a development method. Depending on the answers, we may choose between different domain analysis methods and we can know what kind of components we should find and use.

HDA is used to support compositional reuse (see [5, 3]). The components we want to find are business components. A business component represents appropriate business concept in the information system, and it gives some relevant contribution to the organization [7].

The integration of HDA into Tieto Object must be dealt with as well. When designing a method, or when integrating a technique into an existing method, it is known as local method development [20]. Tolvanen has distinguished five steps that an organization may consider while developing methods in-house. The steps are:

1. Selection of methods
2. Method construction
3. Tool selection and adaptation
4. Introduction of methods
5. Method use

Steps are not mandatory in that one can omit some of them. Further, one can retract any of the steps. Basically one can iterate between steps as necessary.

The first step selects methods and techniques that an organization wants to follow and use [20]. TietoEnator decided to use Hierarchical Domain Analysis as part of Tieto Object. Here it was obvious that HDA should be only one of the techniques that are available for developers and if reuse is not the objective for the development effort it may be omitted.

The second step composes the selected methods and techniques to meet specific objectives of ISD [20]. We linked HDA with Tieto Object and defined their relationships. HDA can use information created in the Business Process Reengineering phase. HDA produces usable information for subsequent phases of the development process. Information is obtained in documents. Document can be organization, project, or component specific.

Organization-specific documentation originates from the first phase of HDA. It describes company-wide domain and gives information about the hierarchy of sub-domains. Also, the description of the company-wide domain serves as a holding place for the identified components. The phases from two to four produce project- and component-specific information. The project-specific documentation contains information about the relationships and communication of the components. The component-specific information describes the component's interface, content, and adaptation.

The third step in the local method development selects and adapts tools for the use. We selected Rational Rose because HDA was presented with UML and TietoEnator uses Rational Rose as one of their CASE tools. Although it was argued that UML is not necessarily the best description language [9], this selection led us back to define some features of the technique and documentation of the results (this point is further elaborated in following sub-section).

The fourth step of the local method development introduces newly developed method into an organization, and in the fifth step we use the method in an organization. Both of these steps were seen through during our pilot project. These steps are described in more detail in the following sub-section.

7.3.2 Results Applying Hierarchical Domain Analysis

This sub-section describes our experience in using HDA technique. We describe each phase of HDA in turn in the way we found it best to perform them during the pilot project. After every phase we point out the differences between HDA in this study and HDA as presented in [9], see also Section 2.3. At the end of each phase we report the impact of the tool usage.

Phase One

The first phase of HDA determines the company hierarchy. As it was argued in Section 2.3, HDA presents hierarchy from the point of view of the organization, i.e., does organization group its units by function performed or by market served? In the case of TietoEnator this is not straightforward. TietoEnator has six key business areas:

1. Finance Sector
2. Services
3. Public Sector
4. Process & Manufacturing
5. Processing & Network Support
6. Application Services

Further, every business area provides a number of services and also serves specific customers. In the case of the Public Sector business area, the services (or function performed, in Mintzberg's vocabulary) are:

1. Development
2. Integration
3. Business related consulting
4. Software
5. Maintenance and support

Public Sector serves also some specified customers, i.e., business areas (or market served, in Mintzberg's vocabulary). The customers are:

1. Government
2. Local authorities
3. Health care

We had a number of options to start modeling our overall hierarchy for the TietoEnator. Among possible solutions were modeling each of TietoEnator's key business area as a domain, modeling each service inside business area or modeling each customer as a domain. After a careful analysis of the problem we chose to model customer as a domain. The main reasons for this were:

1. AKE has outsourced development work to TietoEnator so TietoEnator is seen as AKEs software department.
2. AKE invests to find components to use in its environment.

We modeled the overall domain that includes all the software TietoEnator will make for AKE. Business Process Reengineering phase took place prior to the domain analysis so that we could use its results to determine AKE's functions and to find outhow AKE perceives its software.

AKE's structure supports the services it produces for the community, i.e. this is a market served based structure. Also, software is explicitly mentioned to support the processes AKE performs. Based on these facts we produced hierarchical domain model (Figure 7.1). The results of this phase can be used later on when components are organized for other projects in the domain.

This phase does not need refinement. We are waiting eagerly to see how any ensuing development efforts may alter the model.

We used Rational Rose's package diagram to present organizational structure in a tree-like form. The lower levels of the structure are dependent on the higher levels and they are part of the higher level. Each package is an organizational unit stereotype (strawman inside a circle in Figure 7.1). Further, each package is described with English text so that the reader gets a general idea about the services the organizational unit gives. In this way, knowledge can be spread out in the organization.

Phase Two

Here we define into which sub-domain the current project belongs to and we set boundaries for it. Also, we determine what kind of software systems are needed inside that sub-domain to support it. This proved to be an easy task in contrast to the opinion of other researchers (see e.g., [1, 2]). After picking up the sub-domain we refined it according to the results from Business Process Reengineering.

This phase of HDA was not altered. However, we asked ourselves whether it should be a phase of its own. We kept it as a phase because we believe that boundary setting is an important and in this way its importance is emphasized.

Anyway, it is not useful to formally inspect results from this phase because inspection can be performed as part of the previous or the following phase.

Again we used package diagrams to depict the software systems needed and to show how they relate to each other. This time, however, we had a different point of view of the package. When, in the first phase, the package depicts an organizational unit (see the stereotype in the Figure 7.1) here the package depicts software needs in the chosen domain. One of the software systems most likely appears as the system we are currently working with.

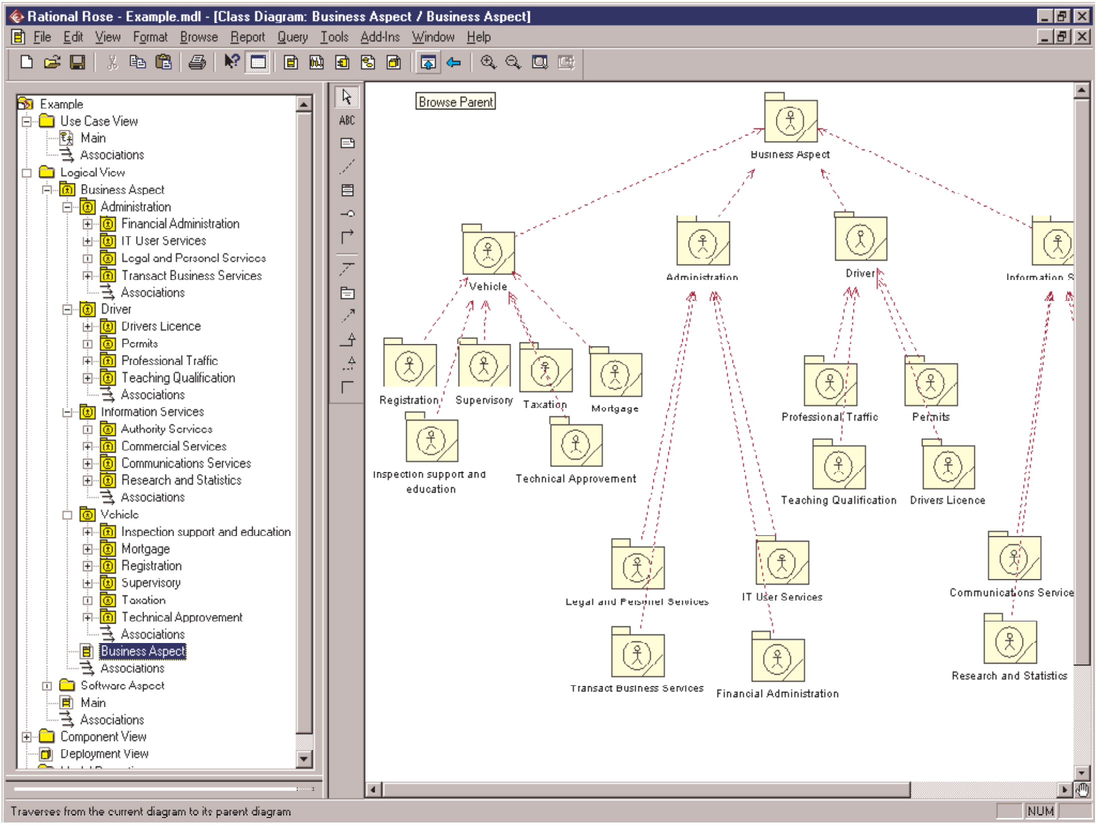


FIGURE 7.1 Resulting Hierarchy from the First Phase of HDA (partial) and Example of Tool Usage.

Phase Three

At this point we have defined the sub-domain where we belong to and our project is most likely to be one of the identified software systems. Now we move our focus inside that software system and look for services it must provide for the users of the software. Services identified are the starting point in defining useful components (Example of a resulting component is in Figure 7.2).

We use the following steps in order to find components that are useful outside our sub-domain:

1. Choose one specific area inside the sub-domain

2. Find services in that area

3. Figure out whether it is general enough to be useful outside the area (e.g. sub-domain specific)

4. Determine boundaries for the service, i.e., what it will solve (general) and what is left open for refinement in realizations of it (variation points)

5. Define the interface (how it is used from outside)

6. Create a class diagram (or other illustrative models)

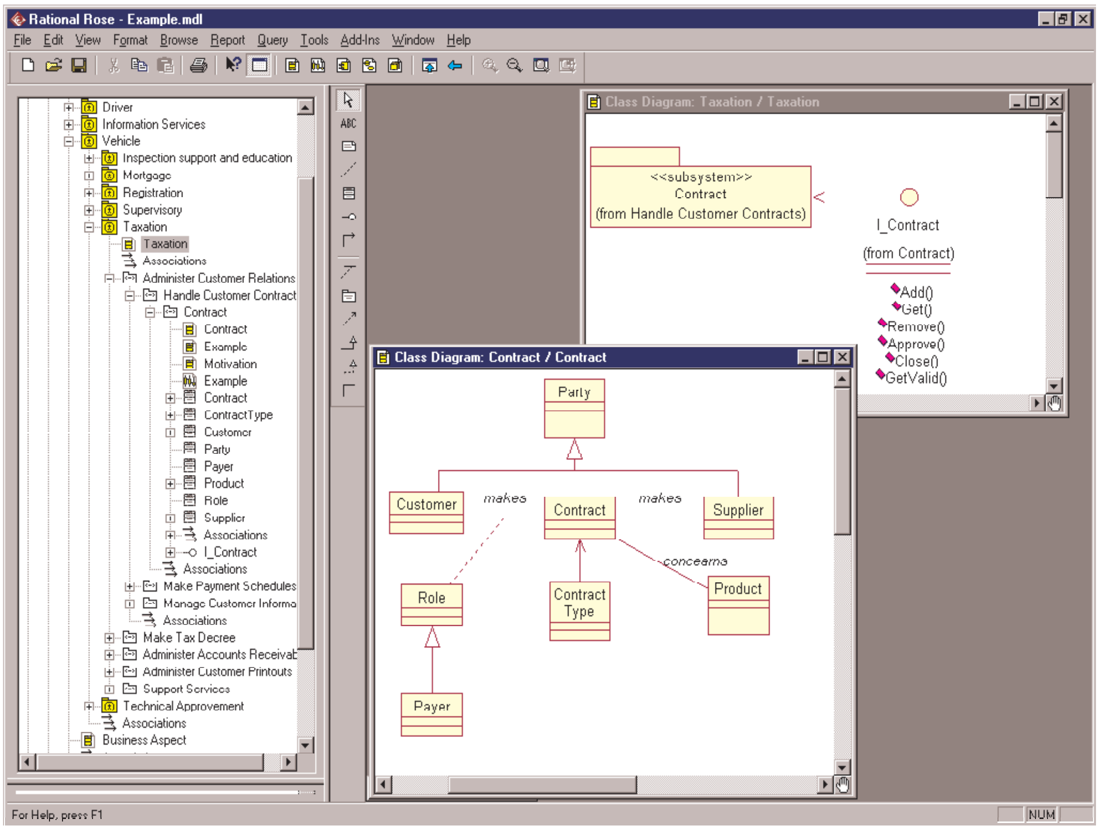


FIGURE 7.2 Example of One Defined Component with Rational Rose.

The first step implies that we should divide our software system into smaller and more manageable areas. After this we analyze each area more closely in turn. From the area chosen we define the services users of the software expect to have. As the result from the second step we have a list of services provided in a defined area of the software system. In the third step we figure out if an identified service is needed outside our software system. If this is the case, we have found a candidate component. The fourth step analyzes each candidate component from the perspective of other sub-domains. The objective here is to find out the general service that can be applied in all sub-domains. Further, we must define what kind of variations each sub-domain needs in order to define the variation points of the component. Based on the results from the fourth step we define the interface for the component (step 5) and the contents of the component (step 6).

The defined components are located into organization hierarchy inside a sub-domain under which the reuse opportunities for the component can be identified. It should be noted that organization hierarchy serves as a holding point for the components, i.e., other project can browse the organization hierarchy and see the components, that are identified as useful. Of course, it is obligatory to use the components found.

This phase was most altered from the point of view of our starting point. It seemed clear to us that if the components should be business components there would be no need for all of the tasks originally proposed in HDA. One reason for this is that the original model was biased towards the Domain Specific Software Architecture (DSSA) [21] technique.

We began to use Rational Rose after we had identified a candidate component. The candidate component was first modeled with Use Cases. After this we depicted the component as a package and decided what kind of services it should offer. This involves first drawing a package diagram and its interface. Then each component package opens to a class diagram where general solution is presented (See Figure 7.2). Finally, there exist one or more examples on how to adapt the component.

Documentation of the component follows design pattern documentation (see [11]). We created our own script for Rational Rose so that documentation could be generated from the diagrams.

Phase Four

Here we determine the components in the middleware and system software layers. This phase is quite straightforward. Because we stayed at the level of defining business components, roughly analysis level results, we concluded that these results should be used only in the subsequent phases of the overall project, namely at the design and implementation phases. Results from this phase are used at the design level of the project to define how components may interact with each other.

This phase does not need enhancements.

Here we use Rational Rose and its package diagram to depict hierarchy in the software aspect of HDA (Figure 7.3). Package diagrams describe software systems that are used, and diagrams show connections between systems.

Phase Five

As we noted in Section 2.3 this is not a phase exactly, but we want to point out that using the found and defined components is the key in reuse. The defined

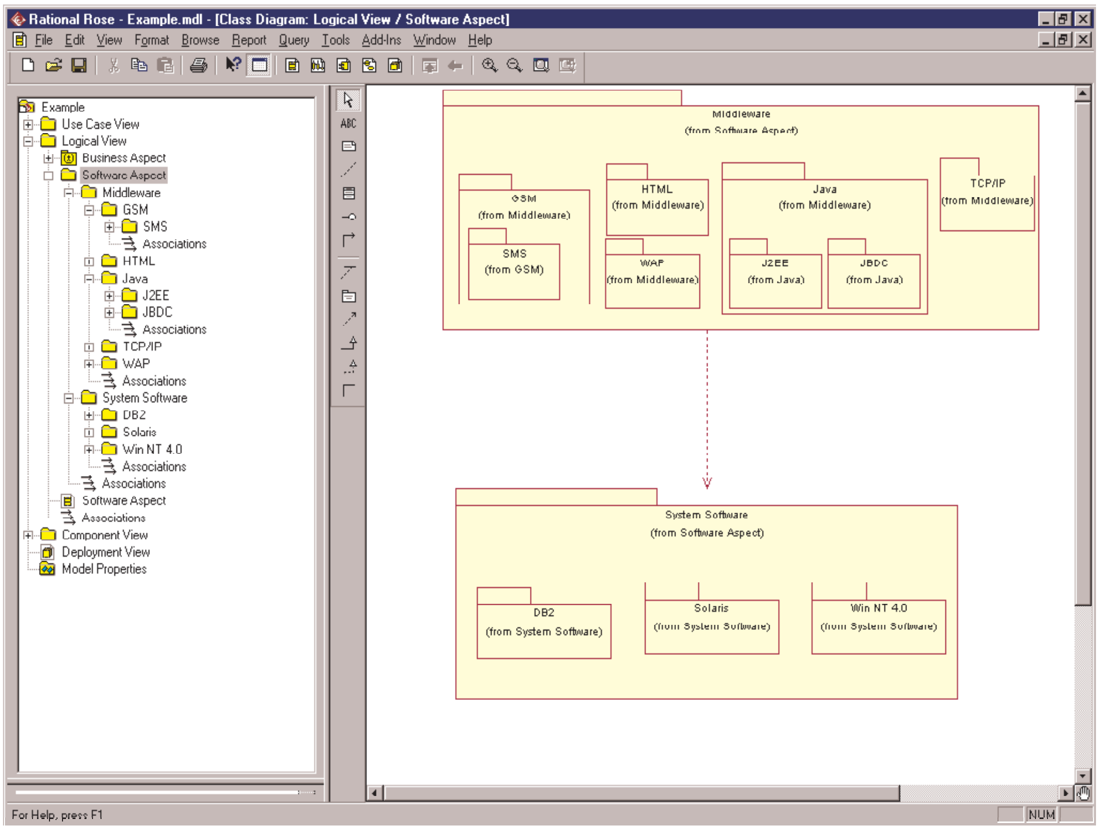


FIGURE 7.3 Sample Screen Shot from Software Aspect.

business components are roughly at an analysis level. As such they point at solutions to some identified problems. These components are used in the analysis and design phases of the software project. Because we also give examples of how to adapt these components for use, incorporating them should be easier. The third phase of HDA tells us where the sub-domains might be useful.

7.3.3 Summary

Method construction and utilization can also be seen as a change in organization. This is why we reflect our experiences through the Leavitt's [14] diamond (Figure 7.4). According to Leavitt [14] one can view organization as complex systems that have at least four interacting variables: task, structure, technology, and people. The task is the reason why an organization exists. It includes the production of goods and services and large numbers of operationally meaningful subtasks. The actors mainly refers to people. The structure includes the systems of communication, authority and work flows. The technology refers to direct problem solving inventions, for example, measurement techniques or computers. Both machines and programs are in this category. Each of the element is highly dependent on the other three. Change in one of the elements usually results in a change in the others.

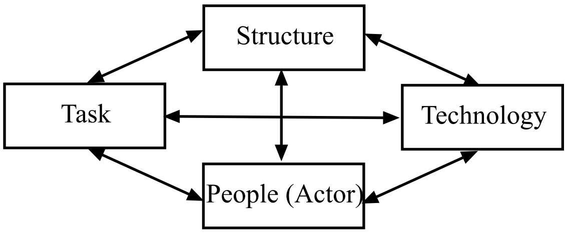


FIGURE 7.4 Leavitt's diamond.

We elaborate each of these components first from the point of view of method construction and secondly, from the point of view of the experience using HDA.

Method Construction

During the method construction we did not define the objectives for reuse. Also, we did not specify what kind of components we want to create with HDA. We first tried using Tracz DSSA [21] for the task. This turned out to be a mistake. Only when we started to apply HDA in the pilot project we noticed our flaw. We then had to come back to define our objectives for reuse and to specify the components to use. This resulted in the redesign of our HDA and in the integration of it into Tieto Object.

Reuse processes should define the creation, brokering and use of components [15]. HDA aids the use and creation of components. Thus there should exist process for brokering components. TietoEnator has defined a brokering process that is repository process in their vocabulary, so we could rely on this. Also, we need materials to teach the newly developed method in the organization. Thus, one part of the pilot project should be the creation of course material. In the method construction we need at least three types of expertise: people who know the development method, people who know the domain analysis technique and people who know about the method construction. Naturally we have to have the support of the top management in order to get the resources needed.

From the technological point of view we need an organization-wide repository where the components found are maintained. We also have to define documentation that is needed in the organization and project levels.

Method Use

The purpose of the pilot project was twofold. First of all we set out to prove the practicality of the technique in finding and using components. Secondly, we were to create course materials to disseminate the technique within TietoEnator. Both of these requirements were fulfilled. We enhanced the technique to make it useful for TietoEnator. We found number of usable business components that can be used more widely inside the domain. Within this pilot project we refined three of them: Contract, Job Queue, and Sales Ledger. Also, we were able to create course material which includes the process description of HDA and examples on its use. One point to ponder is the cost of a developed component. People most likely to use HDA are experts in the domain area and skilled developers. When these people create a reusable component, it is done faster by them than by an average developer and, most likely, the quality of the work is also better. The question arises: if a less skilled developer creates a one-time solution, is the process faster overall?

In HDA, we need people to fulfill three different roles: a domain analyzer, a domain expert, and a steering committee. Persons who have knowledge about the domain are called domain analyzers. We also need developers who closely interact with the actual environment so that they can comment on component requirements from the point of view of technology. A steering committee can validate components. It should include people from each major sub-domain so that it could also point out if a candidate component would be useful in other sub-domains.

The structure of HDA, i.e., the defined phases, proved practical. The second phase of HDA, i.e., defining the sub-domain, was easy to conduct. We believe that this is due to the fact that we use organization's structure to find subdomains. When we have to define boundaries there is an expert from the target organization who can tell whether something belongs to the domain or not. Further, sub-domain, most likely, has already been defined in the organization.

It is interesting to notice that the third phase of HDA involves two levels of refinement. The first one deals with the project level data and the second one with component level data. We found it quite useful to do two kinds of descriptions about the component. First, we described what the component does and rationalized design decisions into these descriptions. Second, we showed how to adapt components in the sub-domain we worked with. We expect to have more of these adaptation descriptions when components are used in other sub-domains.

In our choice of technology we ended up with Rational Rose as our CASE tool. As it is often argued, tool selection impacts the work process and resulting documentation (e.g. [14]). We were constantly fighting against the semantics of UML and how it should be employed in our approach. The usual way to use it is to create a package diagram which may include other packages and diagrams. However, a need for semantical representation within our models soon arose.

The tool did not directly support the documentation of the results. We used component documentation model presented by Forsell and Päivärinta [10] where relevant information clusters about component are identified. After defining documentation we had to create a standard format to describe each model within the Rational Rose. Also, we had to create our own scripts to produce documentation from the models presented with Rational Rose.

7.4 Conclusions

We presented here an action case study where hierarchical domain analysis (HDA) was integrated into Tieto Object software development method and then used for a software development project in an industrial setting. HDA aids finding, designing and reusing components in software development. This should ease the workload in those projects and result in high quality software with fewer resources needed. Results of the pilot project show some improvements that can be made to the original HDA method.

We believe that results gained in this project would not have been possible with the original Tieto Object method. In our opinion the pilot project was successful in three major ways. First of all it produced reusable business components. Secondly, it showed that HDA is an approach that is useful. Third, we gained understanding for the refinement of the HDA and we now have experience and material to educate developers inside TietoEnator Corporation.

We want to point out four other contributions this study makes. First, here we have shown how we integrated HDA into Tieto Object. We believe that our comments about the integration should be taken into account whenever practitioners intend to integrate DA into a development process. Developers must first identify any need for reuse and objectives for it. Then one must determine the kinds of components to be used. Only after this, one can select a domain analysis method. We also listed the steps we took during local method development. Second, we showed that with HDA on can find reusable business components and get an idea about where in the domain they can be reused. Third, we presented how to document and model the results. Fourth, we found out what kinds of roles are needed to take advantage of the results.

The results are usable for practitioners who want to create software from reusable components in an organization with domain analysis. Also, development methodologist can use these results when further developing existing domain analysis methods.

Currently HDA is used in other sub-domains with encouraging results. Here we have quite ideal environment to apply HDA. It would be interesting to try it out in more complex environments. Further work is needed in improving documentation and links with repository processes.

Acknowledgments

This research was supported by Tekes Technology Development Centre, Finland, and companies participating in PISKO project. I would also like to point out my gratitude to TietoEnator Corporation and Jaroslaw Skwarek and Ari Hirvonen there especially.

References

1. Arango, G., Domain analysis - from art form to engineering discipline, Proceedings of the 5th International Workshop on Software Specifications and Design, 152-159 (1989).

2. Arango, G., Prieto-Díaz, R., Introduction and overview: domain analysis concepts and research directions. Edited by Prieto-Díaz, R., Arango, G., Domain Analysis and Software Systems Modeling (IEEE Computer Society Press, Los Alamitos, California, 1991).

3. Arango, G., Domain analysis methods. Edited by Schäfer, W., Prieto-Díaz, R., Matsumoto, M. Software Reusability, 17-49, (Ellis Horwood Ltd., 1994),.

4. Basili, V., Caldiera, G., Cantone, G., A reference architecture for the component factory, ACM Transactions on Software Engineering and Methodology, Vol. 1, No. 1, 53-80 (January 1992).

5. Biggerstaff, T., Richter, C., Reusability framework, assessment, and directions, IEEE Software, 41-49 (March 1987).

6. D'Souza, D., Wills, A., Objects, Components, and Frameworks with UML: The Catalysis Approach (Addison-Wesley, 1999).

7. Eriksson, H-E., Penker, M., Business Modeling with UML: Business Patterns at Work (John Wiley & Sons, Inc., Reading, Massachusetts, 2000).

8. Fitzgerald, B., The use of systems development methodologies in practice: a field study, Information Systems Journal, Vol. 7, No. 3, 201-212 (1997).

9. Forsell, M., Using hierarchies to adapt domain analysis in software development. Ninth International Conference on Information Systems Development, ISD2000, Kristiansand, Norway, 14-16 August 2000 (To be published by Kluwer / Plenum Press at Spring 2001).

10. Forsell, M., Päivärinta, T., A model to document reusable software components. Sent for refereeing. 2001.

11. Gamma, E., Helm, R., Johnsson, R., Vlissides, J., Design Patterns: Elements of Reusable Object-Oriented Software (Addison-Wesley Publishing Company, Reading Massachusetts, 1995).

12. Jaaksi, A., Aalto, J-M., Aalto, A., Vättö, K., Tried & True Object Development: Industry-Proven Approaches with UML (Cambridge University Press, 1999).

13. Jacobson, I., Booch, G., Rumbaugh, J., The Unified Software Development Process (Addison-Wesley, 1999).

14. Leavitt, H., Applied Organizational change in industry: structural, technological and humanistic approaches. Edited by March, J., Handbook of Organizations (Rand McNally & Company, 1965, 3rd printing 1970).

15. Lim, W., Managing Software Reuse (Prentice Hall PTR, Upper Saddle River, NJ, 1998).

16. Mathiassen, L., Reflective systems development, Scandinavian Journal of Information Systems, Vol. 10, No. 1 & 2, 67-118, (1998).

17. Mintzberg, H., Structure in Fives: Designing Effective Organizations (Prentice-Hall, Inc., Englewood Cliffs, N.J., 1983).

18. Nunamaker, J., Chen, M., Purdin, T., Systems development in information systems research, Journal of Management Information Systems, Vol. 7, No. 3, 89-106, (1991).

19. Prieto-Díaz, R., Domain analysis for reusability, Proceedings of COMPSAC'87, 23-29, (1987).

20. Tolvanen, J-P., Incremental Method Engineering with Modeling Tools: Theoretical Principles and Empirical Evidence (Ph.D. Thesis, University of Jyväskylä, Jyväskylä University Printing House, Jyväskylä and ER-Paino Ky, Lievestuore, 1998).

21. Tracz, W., Confessions of a Used Program Salesman: Institutionalizing Software Reuse (Addison-Wesley Publishing Company, 1995).

22. Vidgen, R., Braa, K., Balancing interpretation and intervention in information system research: the action case approach. Edited by Lee, A., Liebenau, J., DeGross, J., Information Systems and Qualitative Research (IFIP, Chapman & Hall, London, 1997).

23. Wartik, S., Prieto-Díaz, R., Criteria for comparing reuse-oriented domain analysis approaches. International Journal of Software Engineering and Knowledge Engineering, Vol. 2, No. 3, 403-432, (September 1992).

YHTEENVETO (FINNISH SUMMARY)

Ohjelmistotuotanto pyrkii tuottamaan laadukkaita ohjelmistoja mahdollisimman kustannustehokkaasti. Yksi tapa tehostaa ohjelmistojen kehittämistä on uudelleenkäyttää aiemmin tuotettuja ohjelmistokomponentteja. Tämä tutkimus osoittaa keinoja, joilla komponenttien uudelleenkäyttöä voidaan ohjelmistoyrityksissä tehostaa.

Tutkimuksen aluksi perehdytään ohjelmistotuotannon nykytilaan tieteellisten julkaisujen perusteella sekä tekemällä ohjelmistoprosessikartoituksia ohjelmistotuotantoa harjoittavissa yrityksissä. Näissä tutkimusmenetelminä käytetään kirjallisuuskatsauksia sekä yrityksissä suoritettavia tapaustutkimuksia. Näiden tutkimusten perusteella voitiin osoittaa ongelmia kohdealueanalyysin käytössä ohjelmistotuotannossa sekä komponenttien dokumentoinnissa. Seuraavaksi tutkimuksessa suunnitellaan ratkaisuja löydettyihin ongelmiin. Tässä vaiheessa hyödynnettiin käsitteellisteoreettista tutkimusta, jonka tuloksena kehitettiin hierarkkinen kohdealueanalyysimenetelmä sekä kehys komponenttien dokumentoinnille. Viimeiseksi esitetään tapaustutkimus, jossa kehitettyjä malleja kokeiltiin käytännössä ohjelmistoyrityksessä.

Tutkimuksen tuloksista voidaan nostaa esiin neljä keskeistä tulosta. Ensinnäkin tutkimuksessa vertaillaan keskenään kolmea komponenttipohjaista ohjelmistonkehitysmenetelmää, joista arvioidaan kuinka hyvin ne tukevat komponenttien uudelleenkäyttöön liittyviä toimenpiteitä. Toiseksi tutkimuksessa esitetään yksinkertainen mutta tehokas tapa mallintaa ohjelmistoprosesseja. Prosessien mallintamistekniikka on kevyt, mutta kuitenkin tuottaa varsin hyviä tuloksia. Kolmanneksi tutkimuksessa esitetään kohdealueen mallintamistekniikka, joka voidaan liittää osaksi yrityksen olemassa olevaa ohjelmistonkehitysmenetelmää. Neljänneksi esitetään tapa dokumentoida komponentit niin, että dokumentointi tukee uudelleenkäyttöä. Tutkimuksen tulokset ovat hyödyllisiä ohjelmistoyrityksille, jotka haluavat kehittää omaa ohjelmistoprosessiaan tukemaan komponenttien uudelleenkäyttöä.

JYVÄSKYLÄ STUDIES IN COMPUTING

1 Ropponen, Janne, Software risk management - foundations, principles and empirical findings. 273 p. Yhteenveto 1 p. 1999.

2 Kuzmin, Dmitri, Numerical simulation of reactive bubbly flows. 110 p. Yhteenveto 1 p. 1999.

3 Karsten, Helena, Weaving tapestry: collaborative information technology and organisational change. 266 p. Yhteenveto 3 p. 2000.

4 Koskinen, Jussi, Automated transient hypertext support for software maintenance. 98 p. (250 p.) Yhteenveto 1 p. 2000.

5 Ristaniemi, Tapani, Synchronization and blind signal processing in CDMA systems. - Synkronointi ja sokea signaalinkäsittely CDMA järjestelmässä. 112 p. Yhteenveto 1 p. 2000.

6 Laitinen, Mika, Mathematical modelling of conductive-radiative heat transfer. 20 p. (108 p.) Yhteenveto 1 p. 2000.

7 Koskinen, Minna, Process metamodelling. Conceptual foundations and application. 213 p. Yhteenveto 1 p. 2000.

8 Smolianski, Anton, Numerical modeling of two-fluid interfacial flows. 109 p. Yhteenveto 1 p. 2001.

9 Nahar, Nazmun, Information technology supported technology transfer process. A multi-site case study of high-tech enterprises. 377 p. Yhteenveto 3 p. 2001.

10 Fomin, Vladislav V., The process of standard making. The case of cellular mobile telephony. - Standardin kehittämisen prosessi. Tapaustutkimus solukkoverkkoon perustuvasta matkapuhelintekniikasta. 107 p. (208 p.) Yhteenveto 1 p. 2001.

11 Päivärinta, Tero, A genre-based approach to developing electronic document management in the organization. 190 p. Yhteenveto 1 p. 2001.

12 Häkkinen, Erkki, Design, implementation and evaluation of neural data analysis environment. 229 p. Yhteenveto 1 p. 2001.

13 Hirvonen, Kullervo, Towards better employment using adaptive control of labour costs of an enterprise. 118 p. Yhteenveto 4 p. 2001.

14 Majava, Kirsi, Optimization-based techniques for image restoration. 27 p. (142 p.) Yhteenveto 1 p. 2001.

15 Saarinen, Kari, Near infra-red measurement based control system for thermo-mechanical refiners. 84 p. (186 p.) Yhteenveto 1 p. 2001.

16 Forsell, Marko, Improving component reuse in software development. 169 p. Yhteenveto 1 p. 2002.