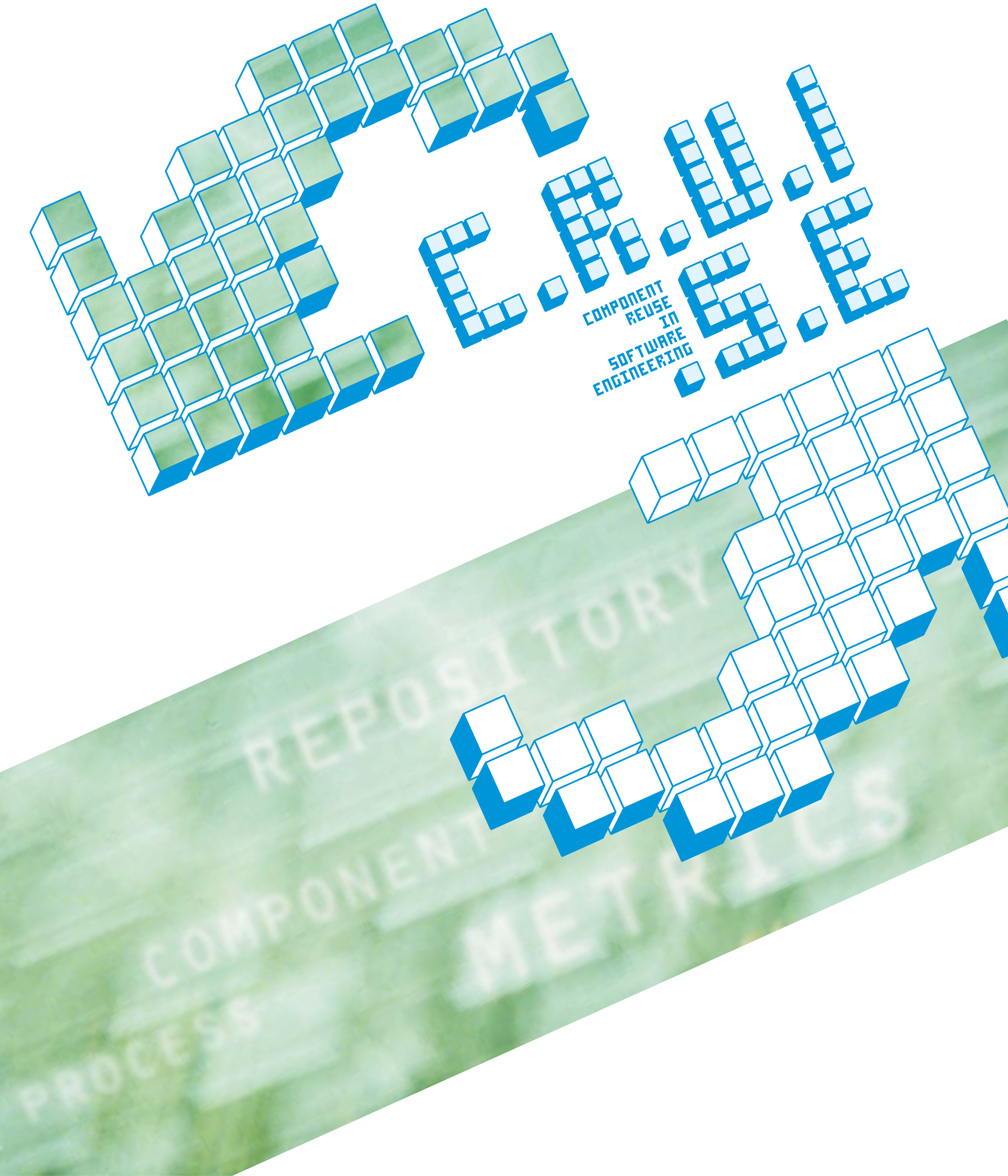
RiSE

c.e.s.a.r e-books

c.e.s.a.r

C.R.U.I.S.E

COMPONENT REUSE IN SOFTWARE ENGINEERING



C.R.U.I.S.E

Component Reuse in Software Engineering

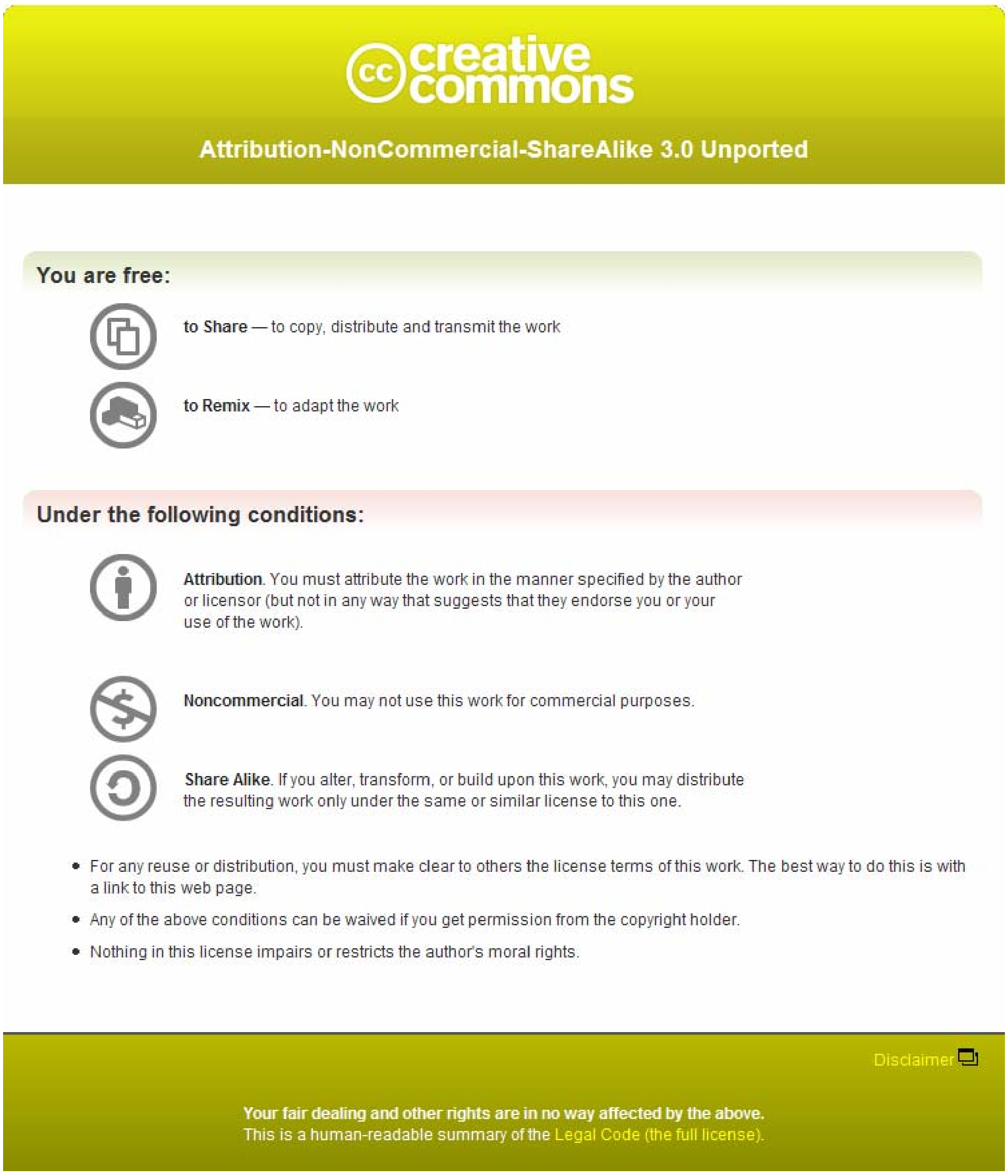
Eduardo Santana de Almeida ● Alexandre Alvaro ● Vinicius Cardoso Garcia ● Jorge Cláudio Cordeiro Pires Mascena ● Vanilson André de Arruda Burégio ● Leandro Marques do Nascimento ● Daniel Lucrédio ● Silvio Lemos Meira

C.R.U.I.S.E.

Component Reuse In Software Engineering

With 19 Figures and 10 Tables

Licensing



creative commons

Attribution-NonCommercial-ShareAlike 3.0 Unported

You are free:

to Share — to copy, distribute and transmit the work

to Remix — to adapt the work

Under the following conditions:

Attribution. You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

Noncommercial. You may not use this work for commercial purposes.

Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under the same or similar license to this one.

- For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to this web page.
- Any of the above conditions can be waived if you get permission from the copyright holder.
- Nothing in this license impairs or restricts the author's moral rights.

Disclaimer

Your fair dealing and other rights are in no way affected by the above.
This is a human-readable summary of the Legal Code (the full license).

Preface

Welcome to Component Reuse in Software Engineering (C.R.U.I.S.E.), a product of the Reuse in Software Engineering (RiSE) Group. RiSE has taken on the formidable task of studying and establishing a robust framework for software reuse, and this book is a milestone achievement in that mission. The following pages contain an extremely thorough review and analysis of the field of reuse by collecting and organizing the major works into a cogent picture of the state-of-our-art.

However, this book is much more than just a very thorough compendium of reuse research. C.R.U.I.S.E actually traces the *history* of our field from the nascent thoughts credited to McIlroy in 1968 right up to the present. In taking the reader through this history, C.R.U.I.S.E touches on all the key phases in the evolution of reuse, from library systems, organizational issues, domain analysis, product lines, component-based engineering, to modern architectures for reuse. In each phase, the leaders in our field have poised numerous concerns and problems that remain to be solved. C.R.U.I.S.E has retained the questions as posed by the original researchers, but also adds new analysis and a current perspective. While it is impossible to cite every single source, the authors have very effectively summarized and analyzed the important works, placing them in context along with the other related works at the time.

One thing that C.R.U.I.S.E will immediately impress upon the reader, as it did me, is the amazing amount of literature and accumulated knowledge that exists in reuse. It is refreshing and with a bit of nostalgia that I again review the landmark articles and read about the first large-scale, practical reuse programs at industry-leading companies such as Hewlett-Packard, Motorola, and IBM, where I began my real hands-on experiences in this exciting field. I particularly like the use of timelines throughout the book to show the progression of the issues in these articles and relationship of the major accomplishments.

In addition to this evolutionary progression, C.R.U.I.S.E analyses the omnipresent issues that challenge the successful introduction of reuse into every organization. These issues range from best practices in development processes, component certification, to library system search and retrieval. Of course the book presents metrics for determining both the value of reuse and the level of reuse practiced by an organization. This topic is near and dear to my heart because of the important role that metrics play in software development and even in helping to encourage the practice of reuse. In this regard, C.R.U.I.S.E is equally valuable to the practitioner seeking to institute a reuse program in their company or organization.

C.R.U.I.S.E not only faithfully traverses the history and past issues surrounding reuse, but the book also gives a thorough analysis of modern practices. There is an entire chapter dedicated to Component-Based Software Engineering (CBSE), which I believe is a key technology and mind-set that organizations must adopt if they really want to use assets in more than one place.

Although reuse is, strictly speaking, the use of un-modified software assets, this book also discusses the important relationships between re-engineering and reuse. Re-engineering, often referred to as “white box reuse,” might reduce up-front development costs but, as many of us know, changing any part of a component can often cost more than just starting from scratch. Nonetheless, there are times that modifying an existing component becomes unavoidable. C.R.U.I.S.E nicely summarizes the issues that arise in these situations, to include version control problems, reduced quality, and the proliferation of software that needs to be maintained.

C.R.U.I.S.E is a must-read for students and researchers prior to beginning any new work in the field. Why? Because simply studying the state-of-the-art is not enough to advance the field. To move forward, researchers need to truly understand the string of successes and failures that shaped where the field is today. Likewise, the wise practitioner would have this book handy to understand the rationale behind a “best practice” that might be questioned as he or she rolls out a reuse program for their company. Quite simply, the history of these findings, failures, and partial successes determines the reason why we do things the way we do.

In conclusion, C.R.U.I.S.E makes one message very clear:

Software Reuse depends on systematic processes and tools.

In other words, reuse doesn’t happen by accident. The research described in this book focuses on issues ranging from technology to management and even to human behavior. Each issue has enjoyed the focus of many experts, and with the benefit of lessons and detailed information contained in C.R.U.I.S.E, you will much better understand the systematic processes and tools that you need for success in your organization.

Dr. Jeffrey S. Poulin
jeffrey.poulin@lmco.com
Lockheed Martin Distribution Technologies
Owego, NY

Table of Contents

Chapter 1
Introduction

Since the time that software development started to be discussed within the industry, researchers and practitioners have been searching for methods, techniques and tools that would allow for improvements in costs, time-to-market and quality. Thus, an envisioned scenario was that managers, analysts, architects, developers and testers would avoid performing the same activities over and over, i.e., a set of reusable assets would be used to solve recurring problems. In this way, costs would be decreased, because the time that would have been necessary to repeat an activity could be invested in others relevant tasks.

The final objective envisioned in this scenario was very clear: to make something *once* and to reuse it *several times*. Thus, the people involved with software development started to explore the idea of reuse, in which experience, design, and, primarily, source code, could be copied from its original context to be reused in a new situation. The first efforts rendered successful stories, which helped to spread the principles of software reuse. The benefits, however, were being mostly achieved in an *ad-hoc* or *opportunistic* fashion. It was not clear why reuse was being achieved and, more importantly, how to achieve it again in other scenarios.

What was needed now was something more *systematic* and *repeatable*. Something based on *defined* processes and rules, so-called systematic reuse

Systematic software reuse is a technique that is employed to address the need for the improvement of software development quality and efficiency, without relying on individual initiative or luck! The objective here was still to reuse all forms of proven experience, including products and processes, as well as quality and productivity models. This rather than create everything anew each time. However, the focus was to provide organizations and managers a way to promote reuse with a greater degree of success.

The goal of this book is to present and discuss the state-of-the-art of software reuse. Methods, techniques and tools, whose main goal is to perform systematic software reuse in industrial environment, are described and discussed.

1.1 Readership

The book is for those people who are interested in the principles of software reuse and its related aspects. It is based upon the foundations of software reuse and provides experience and research-based knowledge.

We have written the book for practitioners, software reuse researchers, professors and students alike.

1.2 Book Overview

The book is organized according to our conceptual framework for software reuse, which has been developed and improved based on our research and experience in reuse gained over four years.

Among other important considerations, the book provides answers to the following questions:

- What are the origins of software reuse? What are the factors that contribute towards success or failure? What are the projects and results from industry around the world? What are some directions for future development?
- What are the techniques used to perform it? How are these techniques organized?
- Which software reuse processes exist? How are they organized? What are their strong and weak points?
- How do you measure the benefits related to software reuse? Which kind of metrics should be considered?
- How to achieve quality in reusable assets? How could a reusable asset be certified, following well-defined standards?
- How do you find, search for and retrieve reusable assets?
- What are the tools used in software reuse? Are they useful?

Besides this brief introduction, the book is divided into 10 chapters, as follows:

Chapter 2 surveys the origins of software reuse ideas, the myths around them, examples of successes and failures, industrial experiences, projects involving software reuse around the world and future directions for research.

Chapter 3 describes the main concepts one of the techniques that promote software reuse - Component-Based Software Development.

Chapter 4 presents eleven processes that represent the state-of-the-art of software reuse and discusses the requirements and other important questions that an effective reuse process must consider.

Chapter 5 analyses the main research into reverse engineering and software reengineering, and discusses how to form an approach towards performing software-reengineering efficiently, aiming towards a larger level of reuse, in order to support further software evolutions. This chapter also presents and discusses some requirements that comprise all issues covered by the research carried out since the 80's.

Chapter 6 shows the relevance of reuse metrics and presents the most common metrics defined for measuring software reuse.

Chapter 7 surveys the state-of-the-art of software component certification. The failure cases that can be found in the literature are also described in this chapter.

Chapter 8 presents a discussion of the role played by reuse repositories in a reuse process; it also shows the main kind of solutions that exist today; and finally, it presents a set of desired requirements for a reuse repository.

Chapter 9 presents a brief discussion of component markets, and a survey of the literature on component searching. Finally, it presents requirements for efficient component retrieval in such markets.

Finally, in Chapter 10 a brief conclusion is given.

1.3 Share Your Experience

We are very interested in your feedback. If you have any criticisms or suggestions for improvements, or if you detected an error or an important issue that the book does not cover, please do not hesitate to contact us at:

rise@cin.ufpe.br

1.4 Acknowledgements

We would like to thank the members of the RiSE (Reuse in Software Engineering) group, for important discussions and seminars about software reuse, and all the other public and private authorities for funding the projects MARACATU (2005), GCOMP (2005-2006), COMPGOV (2005-2007), B.A.R.T (2005-2009), ToolDAy (2006-2009). Most of the results presented in this book have been researched and validated in these projects.

Our special thanks go to the Recife Center for Advanced Studies and Systems (C.E.S.A.R.) who helped us significantly to create and in supporting the RiSE group and its initial dreams.

Last but not least, our thanks go to our collaborators within the industry and the Federal University of Pernambuco. Without the insight acquired in numerous discussions, this book would never have been possible.

Eduardo Almeida, Alexandre Alvaro, Silvio Lemos Meira

Recife Center for Advanced Studies and Systems (C.E.S.A.R.), Federal University of Pernambuco, Brazil, April, 2007.

Chapter 2

Key Developments in the Field of Software Reuse

The history of software development began in the UK in 1948 (Ezran et al., 2002). In that year, the Manchester “Baby” was the first machine to demonstrate the execution of stored-program instructions. Since that time, there has been a continuous stream of innovations that have pushed forward the frontiers of techniques to improve software development processes. From subroutines in the 1960s through to modules in the 1970s, objects in the 1980s, and components in the 1990, software development has been a story of a continuously ascending spiral of increasing capability and economy battled by increasing complexity. This necessity is a consequence of software projects becoming more complex and uncontrollable, along with problems involving schedules, costs, and failures to meet the requirements defined by the customers.

In this context, properly understood and systematically deployed, reuse offers the opportunity to achieve radical improvements in the existing methods of software production. However, it should not be regarded as a silver bullet (Moore, 2001), in which there is a place to put, search and recover chunks of code.

In this way, this chapter surveys the origins of software reuse ideas, the myths that surround it, successes and failures, industry experiences, projects involving software reuse around the world and future directions for research and development.

2.1 Introduction

Code scavenging, reengineering, code generators, or case tools can contribute to increased productivity in software development. The reuse of life cycle assets, mainly code, is often done in an informal and non-systematic way. But, if done systematically, software reuse can bring many benefits.

In the next sections, we will discuss motivations for and definitions for software reuse.

2.2 Motivations

Hardware engineers have succeeded in developing increasingly complex and powerful systems. On the other hand, it is well-known that hardware engineering cannot be compared to software engineering. However, software engineers are faced with a growing demand for complex and powerful software systems, where new products have to be developed more rapidly and product cycles seem to decrease, at times, to almost nothing. Some advances in software engineering have contributed to increased productivity, such as object-oriented programming, component-based development, domain engineering, and software product lines, among others.

These advances are known ways to achieve software reuse. Some studies into reuse have shown that 40% to 60% of code is reusable from one application to another, 60% of design and code are reusable in business applications, 75% of program functions are common to more than one program, and only 15% of the code found in most systems is unique and new to a specific application (Ezran et al., 2002). According to Mili et al. (Mili et al., 1995) rates of actual and potential reuse range from 15% to 85%. With the maximization of the reuse of tested, certified and organized assets, organizations can obtain improvements in cost, time and quality as will be explained next.

2.3 Definitions

Many different viewpoints exist about the definitions involving software reuse.

For Peter Freeman, reuse is the use of any information which a developer may need in the software creation process (Ezran et al., 2002). Basili & Rombach define software reuse as the use of everything associated with a software project, including knowledge (Basili & Rombach, 1991). For Frakes & Isoda (Frakes & Isoda, 1994) software reuse is defined as the use of engineering knowledge or artifacts from existing systems to build new ones. Tracz considers reuse as the use of software that was designed for reuse (Ezran et al., 2002). According to Ezran et al. (2002), software reuse is the systematic practice of developing software from a stock of building blocks, so that similarities in requirements and/or architecture between applications can be exploited to achieve substantial benefits in productivity, quality and business performance.

In this book, the Krueger's general view of software reuse will be adopted (Krueger, 1992):

"Software reuse is the process of creating software systems from existing software rather than building them from scratch".

2.4 The Origins

The idea of software reuse is not new. In 1968, during the NATO Software Engineering Conference, generally considered the birthplace of the field, the focus was the software crisis – the problem of building large, reliable software systems in a controlled, cost-effective way. From the beginning, software reuse was touted as a mean for overcoming the software crisis. An invited paper at the conference: "*Mass Produced Software Components*" by McIlroy (McIlroy, 1968), ended up being the seminal paper on software reuse. In McIlroy's words: "*the software industry is weakly founded and one aspect of this weakness is the absence of a software component sub-industry*" (pp. 80) a starting point to investigate mass-production techniques in software. In the "mass production techniques", his emphasis is on "techniques" and not in "mass production". In the intermediary decades, conductors of all types of research have continued to use the constructions, cars and their industries as example of reuse. Not very many examples of software reuse appear in software reuse literature, which certainly must have a correlation with the state-of-practice of the area.

At the time, however, McIlroy argued for **standard catalogues** of routines, classified by precision, robustness, time-space performance, size limits, and binding time of parameters; to apply routines in the catalogues to any one of a larger class of often quite different machines; to have confidence in the **quality of the routines**; and, finally, the different types of routines in the catalogue that are similar in purpose to be engineered uniformly, so that two similar routines should be available with similar options and two options of the same routines should be **interchangeable** in situations indifferent to that option.

Among the reasons for the gap between McIlroy's ideas and the current state-of-the-practice are: the software industry lags behind the hardware industry, in terms of manufacturing principles and catalogues of standard parts; cultural change in developers, who always use "to build", instead of "to reuse" (Wrong - "What mechanisms do we build?", Right - "What mechanisms do we reuse?"); the fact that current repository systems often do not consider Domain Engineering or Product Line processes for development of artifacts, and the lack of Software Reuse Assurance processes before publishing artifacts for reuse. Finally, most research results ignore a fundamental warning given by McIlroy: "*To develop a useful inventory, money and talent will be needed. Thus, the whole project is an improbable one for university research*." (pp. 84).

2.5 A Consequence of McIlroy's work: Libraries

Based on McIlroy's ideas, much research has explored the library and repository concepts. In (Mili et al., 1998), Mili et al. discuss about 50 approaches for this problem. Prieto-Diaz & Freeman's (Prieto-Diaz & Freeman, 1987) work has given an important contribution to this area.

The work proposed a facet-based scheme to classify software components. The facet scheme was based on the assumptions that collections of reusable components are very large and growing continuously, and that there are large groups of similar components. In this approach, a limited number of characteristics (facets) that a component may have are defined. According to Prieto-Diaz & Freeman, facets are sometimes considered as perspectives, viewpoints, or dimensions of a particular domain. Then, a limited set of possible keywords are associated to each facet. To describe a component, one or more keywords are chosen for each facet.

In this way, it is possible to describe components according to their different characteristics. Unlike the traditional hierarchical classifications, where a single node from a tree-based scheme is chosen, facet-based classification allows multiple keywords to be associated to a single facet, reducing the chances of ambiguity and duplication.

Using this approach, Prieto-Diaz & Freeman evaluated their proposal based upon: retrieval, classification and reuse-effort estimation – as follows:

1. **Retrieval**. The retrieval effectiveness has been tested by comparing recall and precision values of their system to those of a database retrieval system with data not organized by a classification scheme. The result have shown

that recall has been reduced by more than 50 percent, and precision improved by more than 100 percent;

2. **Classification**. The classification scheme has been tested for ease of use, accuracy and consistency. A set of five programs – the faceted schedules, some classification rules, and an example of how to classify a program have been given to 13 graduate computer-science students. The researchers asked the participants to classify each program and to comment about any difficulties experienced during the process. For accuracy and consistency, they have been compared the classifications returned results. Consistency was 100 percent for terms selected from the function and objects facet, and 60 percent for terms from the medium facet; and
3. **Reuse effort estimation**. Prieto-Diaz & Freeman have been asked reusers to rank functionally equivalent components. Next, they have been compared their ranking to the system ranking. With a target application and some implementation constraints, they gave five candidate programs and their respective detailed documentation to six participants. For three of the five programs, ranking correlations were 100 percent - the participants and the system ranked the components in the same order. However, for the relative small size of the collection and limited number of participants, the results were indicative, not conclusive.

2.6 The Benefits

As discussed previously, software reuse has a positive impact on software quality, as well as on costs and productivity (Sametinger, 1997).

Quality Improvements. Software reuse results in improvements in quality, productivity and reliability.

- **Quality**. Error fixes accumulate from reuse to reuse. This yields higher quality for a reused component that would be the case for a component that is developed and used only once;

- **Productivity**. A productivity gain is achieved due to the less code that has developed. This results in less testing efforts and also saves analysis and design labor, yielding overall savings in cost; and
- **Reliability**. Using well-tested components increase the reliability of a software system. Moreover, the use of a component in several systems increases the chance of errors being detected and strengthens confidence in that component.

Effort Reduction. Software reuse provides a reduction in redundant work and development time, which yields a shorter time to market.

- **Redundant work and development time**. Developing every system from scratch means redundant development of many parts like requirement specifications, use cases, architecture, etc. This can be avoided when these parts are available as reusable components and can be shared, resulting in less development and less associated time and costs;
- **Time to market**. The success or failure of a software product is often determined by its time to market. Using reusable components can result in a reduction of that time;
- **Documentation**. Although documentation is very important for the maintenance of a system, it is often neglected. Reusing software components reduces the amount of documentation to be written but compounds the importance of what is written. Thus, only the overall structure of the system and newly developed assets have to be documented;
- **Maintenance costs**. Fewer defects can be expected when proven quality components have been used and less maintainability of the system; and
- **Team size**. Some large development teams suffer from a communication overload. Doubling the size of a development team does not result in doubled productivity. If many components can be reused, then software systems can be developed with smaller teams, leading to better communications and increased productivity.

2.7 The Obstacles

Despite the benefits of software reuse, it is not as widely practiced as one might assume. There are some factors that directly or indirectly influence its adoption. These factors can be managerial, organizational, economical, conceptual or technical (Sametinger, 1997).

Managerial and Organizational Obstacles. Reuse is not just a technical problem that has to be solved by software engineers. Thus, management support and adequate organizational structures are equally important. The most common reuse obstacles are:

- **Lack of management support**. Since software reuse causes up-front costs, it cannot be widely achieved in an organization without support of top-level management. Managers have to be informed about initial costs and have to be convinced about expected savings;
- **Project management**. Managing traditional projects is not an easy task, mainly, projects related to software reuse. Making the step to large-scale software reuse has an impact on the whole software life cycle;
- **Inadequate organizational structures**. Organizational structures must consider different needs that arise when explicit, large-scale reuse is being adopted. For example, a separate team can be defined to develop, maintain and certify software components; and
- **Management incentives**. A lack of incentives prohibits managers from letting their developers spend time in making components of a system reusable. Their success is often measured only in the time needed to complete a project. Doing any work beyond that, although beneficial for the company as a whole, diminishes their success. Even when components are reused by accessing software repositories, the gained benefits are only a fraction of what could be achieved by explicit, planned and organized reuse.

Economic Obstacles. Reuse can save money in the long run, but it is not for free. Cost associated with reuse can be (Sametinger, 1997): costs of making something reusable, costs of reusing it, and costs of defining and implementing a reuse process. Reuse requires up-front investments in infrastructure, methodology, training, tools and archives, with payoffs being realized only years later. Developing assets for reuse is more expensive than developing them for single use only. Higher levels of quality, reliability, portability, maintainability, generality and more extensive documentation are necessary. Such increased costs are not justified when a component is used only once.

Conceptual and Technical Obstacles. The technical obstacles for software reuse include issues related to search and recovery components, legacy components and aspects involving adaptation (Sametinger, 1997):

- **Difficulty of finding reusable software**. In order to reuse software components there should exist efficient ways to search and recovery them. Moreover, it is important to have a well-organized repository containing components with some means of accessing it. This will be detailed in Chapters 8 and 9.
- **Non-reusability of found software**. Easy access to existing software does not necessarily increase software reuse. Reusable assets should be carefully specified, designed, implemented, and documented, thus, sometimes, modifying and adapting software can be more expensive than programming the needed functionality from scratch;
- **Legacy components not suitable for reuse**. One known approach for software reuse is to use legacy software. However, simply recovering existing assets from legacy system and trying to reuse them for new developments is not sufficient for systematic reuse. Reengineering can help in extracting reusable components from legacy system, but the efforts needed for understanding and extraction should be considered; and
- **Modification**. It is very difficult to find a component that works exactly in the same way that we want. In this way, modifications are necessary and there should exist ways to determine their effects on the component and its previous verification results.

2.8 The Basic Features

The software reuse area has three key features (Ezran et al., 2002) as will be detailed next.

1. Reuse is a systematic software development practice. Before describing systematic reuse, it is important to consider non-systematic reuse, which is ad hoc, dependent on individual knowledge and initiative, not deployed consistently throughout the organization, and subject to little if any management planning and control. If the parent organization is mature and well managed, it is not impossible for non-systematic reuse to achieve some good results. However, more probable is that non-systematic reuse will be chaotic in its effects, feed the high-risk culture of individual heroics and fire-fighting, and amplify problems and defects rather than dampen them. On the other hand, systematic software reuse means:

- understanding how reuse can contribute toward the goals of the whole business;
- defining a technical and managerial strategy to achieve maximum value from reuse;
- integrating reuse into the total software process, and into the software process improvement programme;
- ensuring all software staff have the necessary competence and motivation;
- establishing appropriate organizational, technical budgetary support; and
- using appropriate measurements to control reuse performance.

2. Reuse exploits similarities in requirements and/or architecture between applications. Opportunities for reuse from one application to another originate in their having similar requirements, or similar architectures, or both. The search for similarities should begin as close as possible to those points of origin – that is, when requirements are identified and architectural decisions are made. The possibilities for exploiting similarities should be maximized by having a development process that is designed and managed so as to give full visibility to the flow down from requirements and architecture to all subsequent work products.

3. Reuse offers substantial benefits in productivity, quality and business performance. Systematic software reuse is a technique that is employed to address the need for improvement of software development quality and efficiency (Krueger, 1992). Quality and productivity could be improved by reusing all forms of proven experience, including products and processes, as well as quality and productivity models.

Productivity could be increased by using existing experience, rather than creating everything from the beginning (Basili et al., 1996). Business performance improvements include lower costs, shorter time to market, and higher customer satisfaction, which have already been noted under the headings of productivity and quality improvements. These benefits can initiate a virtuous circle of higher profitability, growth, competitiveness, increased market share and entry to new markets.

2.9 Reusable Software Assets

Before continuing with issues related to software reuse it is necessary to define a central concept in the area: reusable software assets. According to Ezran et al. (2002), software assets encapsulate business knowledge and are of high value to a company. They are composed of a collection of related software work products that may be reused from one application to another.

There are two kinds of assets: **(i)** vertical assets which are specific to an application domain, for example, financial object models, algorithms, frameworks; and **(ii)** horizontal assets which are easier to identify and reuse because they represent recurrent architectural elements. Horizontal assets can be reused independently of the application domain, with the main constraint being that the application architectural choices must be compatible with the asset. Examples of them include GUI objects, database access libraries, authentication service, and network communication libraries.

Besides of these two kinds, assets may have different sizes and granularities such as a function or a procedure, a class, a group of classes, a framework and an application or a product. A reusable asset is potentially made up of many life-cycle products including: requirements and architecture definition, analysis model, design models and code, test programs, test scenarios and test reports.

2.10 The Company Reports

There are very few reports on software reuse experiences in the practice. In this section, the three main cases found in the literature, IBM, Hewlett-Packard (HP) and Motorola, will be presented.

2.10.1 Software Reuse at IBM

In 1991, the Reuse Technology Support Center was established to coordinate and manage the reuse activities within IBM (Bauer, 1993), (Endres, 1993). One component of the established reuse organization was a Reusable Parts Technology Center in Boeblingen, Germany, with the mission to develop reusable software parts and to advance the state-of-the-art of software reuse.

The history of the Boeblingen parts center dates back to 1981. It started as an advanced technology project looking at methods for reusable design. In the beginning, the goal of the project was to have an integrated software development system that supported the reuse of parts. The following steps present the parts center's evolution.

1. The need of a parts center. According to Bauer (1993), before the existence of a reusable parts center, reuse of code across project borders seldom took place. No organizational structures supported cross-project communication, the lack of a common design language made communication difficult and many different description methods for code were in existence. An attempt was made to introduce a common design language within IBM, but it met with little success.

The code that was written day after day was poor and not designed to be reused; hence, the work of designing, testing, and maintenance could not be reused.

2. Reusable design. Early investigations into reusable design at the IBM showed that common structures existed in the area of data structures, operations, and modules. Thus, to avoid such parallel efforts in the future, a formal software development process could be established where inspection, verification, testing, and documentation of the code would be performed once and then made available to other development groups.

It was expected then, that developers could change their way of writing software so that an additional outcome of each project would be high quality parts that could be supplied to a common database.

The approach to find appropriate parts was to scan existing products and to identify replicated functions, either on the design or on the code level. Soon it became clear that an abundance of dependencies on global data existed. This led to the hypothesis that data abstraction, i.e., bundling the data with their operations, was the key to reusability.

3. Reusability through abstract data types. In 1983, a project was started to explore data abstraction as a software development and reuse technology in a real software product: a network communication program. For this project, seven abstract data types called building blocks were written that represented a third of the total lines of code.

4. The building-block approach. Due to the good experiences with abstract data types, the features that were most important for reuse were summarized in a reuse model, postulating five conditions of reuse: information hiding, modularity, standardization, parameterization, and testing and validation.

The project showed that the programming language (PL/S) primarily used within IBM at the time-lacked features that would support the reuse conditions. Therefore, a language extension based on generic packages of Ada, was envisioned, developed, and validated on a second pilot project with success.

5. Development of the reuse environment. As more and more building blocks were produced, tools became necessary to support the user in finding and integrating the building blocks into programs and the environment. The reuse environment was intended to be the integrated development environment for users of Boeblingen's software building blocks. As such, it supported ordering, shipping, maintenance, and (re) use of building blocks. It also offered comprehensive information about available building blocks and support communication among building-block users and developers (information exchange, problem reporting). Moreover, special tools for creating, compiling, and testing building-block applications were also provided.

At the end, a project was set up that resulted in a comprehensive Boeblingen's catalog of reusable abstract data types written in C++. In August 1991, the first release of the class library (IBM’S Collection Class Library) for C++ became available within IBM, and by the end of 1991, 57 projects with 340 programmers were already using the C++ building blocks. Finally, in June 1993, the Collection Class Library became available as part of the IBM C Set++ compiler, for C++.

2.10.2 Software Reuse at Hewlett-Packard

Martin Griss reports in (Griss, 1994), (Griss, 1995) that reuse efforts were initiated at HP in the early 80’s, with the goal of reducing time-to-market, improving software productivity and improving product consistency. Ten years later, HP began a multi-faceted Corporate Reuse Program in order to gather information about reuse from within HP and other companies.

The corporate reuse program was aimed at packaging best-practice information and guidelines to avoid common pitfalls. Additionally, technology transfer and educational processes to spread information and enhance reuse practices within the company were also implemented. Griss and colleagues learned that, for large-scale reuse to work, the problems that need to be overcome are mostly non-technical, and, in particular, he points out that the three most critical elements to a successful reuse program are: **(i)** management leadership and support, **(ii)** organizational change, and **(iii)** the creation of a reuse mindset.

The HP experience was/is important to remind us of some reuse myths normally accepted as truths by most:

- **Development**: *“Produce that product, and on the way, produce components too”*;
- **Object Technology***: "Reuse requires using object technology";*
- **Library systems**: *"Reuse means large libraries and powerful library-management systems";*
- **Reuse and performance**: *"Reusable components are always slow, due to their generalization";*
- **Reuse and development effort**: *"The need for such optimization makes reuse not worth the development effort";* and
- **Reuse and process maturity**: *"You should not even consider systematic reuse until your organization is at level 3 in the Software Engineering Institute's Capability Maturity Model".*

The vision expressed in the first myth (development) is naive, given that component production is a very complex process, involving several tasks, such as domain analysis, separation of concerns, contracts, tests, packaging and others. Thus, a distinct component production organization is needed, which can take candidate components and carefully package, document and certify them. The other myths (Object Technology, Library systems, Reuse and performance, Reuse and development effort, and Reuse and process maturity) will be discussed in detail in Sections 2.11, 2.12 and 2.13, where research reports on such factors are discussed.

With the maturiry of the corporate reuse program, Griss has developed an interesting incremental model for reuse adoption. According to which, one way to adopt reuse incrementally is to increase the investment and experience in the company, thus, focusing on reuse and learning more about the processes, the level of reuse will increase.

The steps of the model evolve from copy and paste reuse, with all its problems to systematic reuse based upon domain engineering. The key point of this model is to consider the incremental adoption of software reuse and to know that without increasing investment and experience (in everything, from processes to learned lessons) it is very difficult to achieve systematic software reuse.

2.10.3 Software Reuse at Motorola

The Motorola drive to increase quality and productivity, starting around the (early 90's) had software reuse as one of its pillars (Joss, 1994). One of the first problems to be dealt with was changing the software development environment from a hardware to a software, mindset. Once this process was well established, Motorola started a 3-phase software reuse process.

Grass-Roots Beginning, the first phase included the creation of a reuse task force including a leading technician from each sector of the company, to investigate and make recommendations about software reuse in the company. The task force considered the need and requirements for education and motivation, methods, technology, and implementation.

The task force had two main activities: **(i)** to educate how to spread the software reuse knowledge; and, **(ii)** to emphasize metrics, rewards, career paths and job descriptions for users and producers of reusable components. Next, the task force recommended approaching reuse from two perspectives. One involved the design, recovery and subsequent reengineering, in order to recover possibly reusable components from existing requirements, design, and code, and then use them to populate reuse databases. The second approach focused on forward engineering, or design for reuse, appropriating resources to create new, reusable software. The task force leader reported later that the main problem in this phase was a lack of knowledge ("*Our group also suffered from ignorance. As a whole we had little common knowledge about, or expertise, in software reuse*").

A reuse working group, formed by fifteen engineers, replaced the task force and became the focal point for reuse activities, concerning itself primarily with software-reuse education, guidelines, technology, and implementation at Motorola. This group focused on education as the most important activity; without it, the other activities would not come to fruition. However, timing and time wre the main barrier, because some participants were volunteers, and did not work full-time.

The second phase, ***Senior-Management Involvement***, was characterized by the fact that Motorola's middle managers were reluctant to adopt software reuse because of the up-front costs and slow return on investment (three or more years). They allowed the grass-roots effort that initially drove its adoption to become ineffective. Realizing this, senior management accepted the responsability of initiating reuse. Next, George Fisher, Motorola's CEO, took the initiative to get upper management more involved with advancing the state-of-the-practice in software. Still in this phase, the primary problem that the reuse working group had encountered was that software reuse is more a cultural than a technological issue. Software engineers readily adopt software reuse technology, but need sustancial management support because developing reusable software consumes significantly more time than schedules usually allow for. The reuse leader relates that the main benefit of this phase was the upper management's involvement in software engineering activities.

In the last phase, ***Future Markets and Tools***, a few inspired groups have realized correctly that software reuse promises more than internal savings: It provides new markets for reusable components and support tools. Thus, once the organization has implemented a process that embodies software reuse, tools were necessary to facilitate software development. In this context, Motorola had begun the development of a reuse toolkit that allowed design capture, reengineering, and component-based development, in order to achieve systematic reuse. According to Joos, the reuse tookit was scheduled for a prototype in the first quarter of 1996, however, the literature does not present any result related to it.

2.11 Success and Failure Factors in software reuse

Because software reuse is very likely a competitive advantage, there is little incentive to share lessons learned across companies. Moreover, there has been little research to determine if an individual company's software reuse success factors are applicable to other organizations. This section presents and discusses the main factors of success and failure related to software reuse available from the literature, trying to establish a relationship among them.

2.11.1 Informal Research

Over more than a decade, companies wanted to achieve systematic reuse, in a given domain, based on repeatable processes, and were concerned primarily with reuse of higher level lifecycle artifacts, such as requirements, design and code. This section presents some attempts of research aiming towards systematic reuse.

According to Frakes & Isoda (Frakes & Isoda, 1994), a key concept of systematic reuse is the domain, which may be defined as an application area or, more formally, a set of systems that share design decisions. In this context, "*systematic software reuse is a paradigm shift in software engineering, from building single systems to building families of related systems*".

Although the literature describes the benefits of software reuse, there is no cookbook solution for systematic reuse. Each organization must analyze its own needs, implement reuse metrics, define the key benefits it expects, identify and remove impediments, and manage risks (Frakes, 1993).

Frakes & Isoda list six factors that are critical for systematic software reuse: *Management*, *Measurement*, *Legal issues*, *Economics*, *Design for reuse* and *Libraries*.

1. **Management.** Systematic reuse requires long-term, top-down management support because:

- It may require years of investment before it is paid off; and
- It involves changes in organizational funding and management structures that can only be implemented with upper management support and guidance, without which none of the other reuse activities is likely to be successful.

2. **Measurement**. As with any engineering activity, measurement is vital for systematic reuse. The reuse level - the ratio of reused to total components - may involve reusing components developed outside the system (external reuse level) or for the system (internal reuse level). In general, reuse benefits (improved productivity and quality) are a function of the reuse level which, in turn, is a function of reuse factors, the set of issues that can be manipulated to increase reuse, either of managerial, legal, economic as technical background (Frakes, 1993);
3. **Legal issues**. Legal issues, many of them still to be resolved, are also important. For example, what are the rights and responsibilities of providers and consumers of reusable assets? Should a purchase of reusable assets be able to recover damages from a provider if the component fails in a critical application?;
4. **Economics**. Systematic reuse requires a change in software economics. Costs incurred by a project that creates reusable assets right to be recovered from the users of those assets and the organization must be able to work out a cost-benefit analysis in order to determine if systematic reuse is feasible. An initial analysis has shown that some assets must be reused more than dozen times in order to recover development costs (Favaro, 1991). However, about fifteen years after this a more complete study proven that the development costs are recovered after two reuses (Poulin, 1997);
5. **Design for reuse**. Designing for systematic reuse requires processes to be systematic also, and domain engineering is a sheer necessity that involves identifying, constructing, cataloging and disseminating a set of software components can possibly be used in existing or future software, in a particular application domain (Frakes & Isoda, 1994); and

6. **Libraries**. Once an organization acquires reusable assets, it must have a way to store, search, and retrieve them – a reuse library. There is an extensive body of literature on methods and tools for creating reuse libraries, with many representation methods coming from from library science and artificial intelligence, among other areas. Although libraries are a critical factor in systematic software reuse, they are not a necessary condition for success with reuse, as will be shown in the following sections.

How is it that software reuse, an obvious winner, can fail? Card & Comer (Card & Comer, 1994) search for answers based on two fundamental mistakes contributing to failure. First, organizations treat reuse as a technology-acquisition problem instead of a technology-transition problem. Currently, the variability of reuse techniques such as Domain Engineering, Component-Based Development, and Product Lines is now mature enough for industrial use (although some problems remain). However, just buying technology usually does not lead to effective reuse. The second mistake is that even organizations that recognize reuse as a technology-transition issue may fail to address the business implications of reuse, and in consequence, to deal with reuse as a business strategy.

Card & Comer point to the fact that an effective reuse program must be market-driven. Thus, an organization implementing reuse must invest in assets that future users will want. A market insensitive reuse program results in a library of assets of such a poor quality, often implemented in an incompatible language, that consumers do not trust them. The problem is that reuse concerns are typically not found in the business plans of software intensive system providers. This probably means that occasions of large-scale reuse are most likely accidental and not repeatable. Consistently achieving high levels of reuse depends on understanding and taking advantage of the economic dimensions of reuse.

Based on this analysis, Card & Comer present four important factors for an effective software reuse program: *training*, *incentives* and *measurement*, and *management*, also highlighted by Frakes and Isoda (see section above).

Glass (Glass, 1998) presents and discusses the question that software reuse has problems in practice. Glass considers that, if there is a motherpie-and-applehood topic in software engineering, reuse is such topic. His point of view is that software reuse has good potential, but this has not been achieved yet. While other researchers point to factors such as management and culture, Glass believes that the main problem is that there are not many software components that can be reused. Additionally, Glass discusses three important questions related to software reuse:

Speed and Quantity vs. Quality and Reusability. Glass advocates that developments within the software-making business placed further obstacles in the path of effective reuse. As it matured, the field moved from an era when getting a good product out the door mattered most to an era when meeting a sometimes arbitrary schedule dominated. Thus, in practice, few people had time to think about reusable components because, as most experts agree, it takes longer to build something reusable than something merely usable.

Another question pointed to by Glass was that most software measurements focused on quantity. Initially, new products were measured in lines of code, thinking that the more lines written, the more significant the project must be. Reuse, which takes longer and results in a product with fewer new lines of code, bucked that trend. The reward system failed to reward reuse.

These two factors involve management. In this context, Glass agrees with the other researchers who say that management is one of the key points in reuse. Management can ward off schedule pressures and gain time for building reusable artifacts.

Conceptual vs. Component Reuse. Glass highlights that in order to comprehend the software reuse scenario one must also consider conceptual reuse thus, rule architectures, frameworks, and patterns show that not just components can be reused, but other kinds of artifact as well. However, this is not a new approach. Willemien Visser (Visser, 1987) said almost the same thing at almost the same time: "*Designers rarely start from scratch.*" Good software professionals' keep their mental backpacks filled with previously used concepts that they can apply when a familiar-sounding "new" problem comes along.

2.11.2 Empirical Research

There is little empirical research into software reuse. In this section, the three main studies available in the literature are presented.

Rine (Rine, 1997) researched the fact that presently there is no set of success factors which are common across organizations and have some predictable relationship to software reuse.

Once software reuse provides a competitive advantage, there is little incentive to share lessons learned across organizations. There has been little research to determine if an individual organization's software reuse success factors are applicable to other organizations. Moreover, Rine highlights that the majority of current information available on software reuse comes from literature, which contains unproved theories or theory applied in a limited way to a few pilot projects or case studies within individual application domains. This lack of practical results is coherent, once we found fewer practical reports in the literature.

In this way, in 1997, Rine (Rine, 1997) conducted a survey to investigate what are the success factors for software reuse that are applicable across all application domains. The organizations in this study with the highest software reuse capability had the following: *product-line approach*, *architecture which standardizes interfaces and data formats*, *common software architecture across the product-line*, *design for manufacturing approach*, *domain engineering*, *reuse process*, *management which understands reuse issues*, *software reuse advocate(s) in senior management*, *state-of-the-art reuse tools and methods*, *experience in reusing high level software artifacts such as requirements* and *design*, *instead of just code reuse*, and *the ability to trace end-user requirements to the components*.

Rine also criticizes the question that many times a repository system is seen as a necessary condition for obtaining success in software reuse. In his study, seventy percent of his survey’s respondents were attempting to implement an approach based on repositories. For Rine, the vast majority of these practitioners have not achieved the levels of success they expected. The reason for this lack of success is that the repository approach does not resolve interchangeability issues or attend to customer needs prior to adding components to the repository.

It is true that just a repository system alone will not guarantee success in software reuse, however, these systems associated with certified components, stored in specific domains, with efficient mechanisms for search and retrieval, can offer significant results for organizations and users. The lack of repositories with these features is also the major impediment for software reuse adoption, which will be discussed in Section 2.16.

Morisio et al. research (2002), originally the most detailed practical study into software reuse available in the literature, although it started some years ago, present and discuss some of the key factors in adopting or running a company-wide software reuse program. The key factors are derived from empirical evidence of reuse practices, as emerged from a survey of projects for the introduction of reuse in European companies: 24 such projects conducted from 1994 to 1997 were analyzed using structured interviews.

The projects were undertaken in both large and small companies, working in a variety of business domains, and using both object-oriented and procedural development approaches. Most of them produce software with high commonality between applications, and have at least reasonably mature processes. Despite the apparent potential for success, about one-third of the projects failed.

Morisio et al. identified three main causes of failure: *not introducing reuse-specific process, not modifying non-reuse processes, and not considering human factors*. Morisio et al. highlight that the root cause was a lack of commitment by top management, or non-awareness of the importance of these factors, often coupled with the belief that using the object-oriented approach or setting up a repository is all that is necessary to achieve success in reuse.

Conversely, successes were achieved when, given a potential for reuse because of *commonality among applications, management committed to introducing reuse processes, modifying non-reuse processes*, and *addressing human factors*.

In (Rothenberger et al., 2003), Rothenberger et al. investigate the premise that the likelihood of success of software reuse efforts may vary with the reuse strategy employed and, hence, potential reuse adopters must be able to understand reuse strategy alternatives and their implications.

In order to engage the reuse phenomenon at a level closer to the organizational reuse setting, Rothenberger et al. focus on the systematic reuse of software code components. These reuse practices vary by extent of organizational support, integration in the overall development process, and the employed techniques. Depending on the reuse practice, reuse may happen in an unplanned fashion when individual developers recognize reuse opportunities informally or it may be planned for by building and maintaining a software repository that supports the predictable and timely retrieval of reusable artifacts.

In this context, the researchers used survey data collected from 71 software development groups to empirically develop a set of six dimensions that describe the practices employed in reuse programs. The respondents included project managers, software engineers, developers, software development consultants, and software engineering researchers in profit and nonprofit organizations in a variety of industries, including telecommunication, financial services, avionics, government, and education. There were no duplicate participants within a development group and almost 80 percent of the participants were working in organizations with more than 200 employees.

The study concluded that higher levels of reuse program success are associated with *planning*, *formalized processes*, *management support*, *project similarity*, and *common architecture*.

Additionally, the study showed that, although object technologies were initially a candidate for a reuse dimension, they have not been used in the final classification scheme. As it was determined to be insignificant in explaining reuse success, Rothenberger et al. concluded that an organization’s reuse success is not dependent on the use of object-oriented techniques. Nevertheless, object technologies may be conducive to reuse, yet the other dimensions that make up the model ultimately determine reuse success. The qualitative analysis of the clusters’ reuse success yielded additional insights: While an improvement of software quality can be achieved without an emphasis on the reuse process, an organization will only obtain the full benefit of reuse if a formal reuse program is employed and subject to quality control through formal planning and continuous improvement.

In (Selby 2005), Selby presents research whose goal is to discover what factors characterize successful software reuse in large-scale systems. The research approach was to investigate, analyze, and evaluate software reuse empirically by mining software repositories from a NASA software development environment that actively reuses software. In Selby’s study, 25 software systems ranging from 3000 to 112.000 source lines was analyzed. The environment has a reuse average of 32 percent per project, which is the average amount of software either reused or modified from previous systems.

The study identified two categories of factors that characterize successful reuse-based software development of large-scale systems: module design factors and module implementation factors. The module design factors that characterize module reuse without revision were: few calls to other system modules, many calls to utility functions, few input-output parameters, few reads and writes, and many comments. The module implementation factors that characterize module reuse without revision were: low development effort and many assignment statements. The modules reused without revision had the fewest faults, fewest faults per source line, and lowest fault correction effort. The modules reused with major revision had the highest fault correction effort and highest fault isolation effort as well as the most changes, most changes per source line, and highest change correction effort.

2.11.3 Big Problems

During this chapter, the benefits and potentials involving software reuse were presented. However, the area is composed also of a big accident related to it (Jezequel & Meyer, 1997). On June 4, 1996, the maiden flight of the European Ariane 5 launcher crashed, about 40 seconds after takeoff. Media reports indicated that a half-billion dollars was lost. A particular aspect of this case was that the error came from a piece of the software that was not need.

The software involved is part of the Inertial Reference System. Before liftoff, certain computations are performed to align the system. Normally, these computations should cease at 9 seconds, but because there is a chance that a countdown could be put on hold, the engineers gave themselves some leeway. According to Jezequel, the engineers reasoned that, because resetting the SRI could take several hours (at least in earlier versions of Ariane), it was better to let the computation proceed than to stop it and then have to restart it if liftoff was delayed. So the system computation continues for 50 seconds after the start of flight mode-well into the flight period. After takeoff, this computation is useless. In the Ariane 5 flight, however, it caused an exception, which was not caught.

The exception was due to a floatingpoint error during a conversion from a 64-bit floating-point value, representing the flight's "horizontal bias," to a 16-bit signed integer. According to the inquiry board a main lesson learned of this fact was that reuse without a precise, rigorous specification mechanisms is a risk of potentially disastrous proportions.

2.12 Success and Failure Factors: Summary

Table 2.1 and Table 2.2 show, respectively, the relation among the works described in Section 2.11 and the failure and success causes. In Table 2.1, each column corresponds to a failure cause: HF. Not considering human factors, RS. Not introducing reuse-specific processes, NR. Not modifying non-reuse processes, TA. Organizations treat reuse as technology acquisition problem, BS. Organizations fail to approach reuse as a business strategy, OO. Reuse requires OO technology, LS. Reuse requires powerful library systems.

Table 2.1. Relation among the reports and the failure causes.

Author (s)	**HF**	**RS**	**NR**	**TA**	**BS**	**OO**	**LS**
Griss, 1994, 1995						x	x
Card & Comer, 1994				x	x	x	x
Morisio. et. al., 2002	x	x	x	x		x	x
Rothenberger. et. al., 2003						x	

In the Table 2.2, each column is related to a success cause: CA. Common Architecture, CO. Components, DR. Design for reuse, DE. Domain Engineering, EC. Economics, FM. Formal Methods, FP. Formalized Process, HF. Human Factors, IN. Incentives, LI. Legal Issues, LS. Libraries, MN. Management, ME. Measurement, NR. Modify non-reuse process, PN. Planning, PL. Product Lines, PS. Project similarity, RP. Reuse process, RT. Reuse tools, TR. Training.

Table 2.2. Relation among the reports and the success causes.

Author (s)	CA	CO	DR	DE	EC	FM	FP	HF	IN	LI	LS	MN	ME	NR	PN	PL	PS	RP	RT	TR
Bauer, 1993		x	x			x					x	x								x
Enders, 1993		x	x			x					x	x								x
Griss, 1994, 1995							x				x	x								x
Joos, 1994									x			x							x	x
Card & Comer, 1994									x			x	x							x
Frakes & Isoda, 1994			x		x					x	x	x	x							
Rine, 1997	x	x	x	x								x				x		x	x	
Glass, 1998												x								
Morisio et al., 2002								x				x		x						
Rothenberger et al., 2003	x						x					x			x		x			

2.13 Myths, Facts and Fallacies of Software Reuse

Many organizations are implementing systematic reuse programs and need answers to practical questions about reuse. However, as shown in Section 2.11.2., little empirical data has previously been collected to help answer these questions. Frakes & Fox (Frakes & Fox, 1995) have answered sixteen questions that are commonly asked about reuse, using survey data collected from organizations in the U.S. and Europe. The analysis supports some commonly held beliefs about reuse and contradicts some others.

The Approach. Frakes & Fox surveyed software engineers, managers, educators and others in the software development and research community about their attitudes, beliefs, and practices in reusing code and other lifecycle objects. Survey respondents were drawn arbitrarily from the software engineering community. A total of 113 people from 28 U.S. organizations and one European organization responded to the survey during two years. Fifty percent of respondents had degrees in computer science and information science.

Questions and answers. The questions chosen are commonly asked by organizations attempting to implement systematic reuse. Table 2.3 summarizes the sixteen questions about reuse based on the survey.

Table 2.3. Frakes & Fox sixteen questions and answers (Frakes & Fox, 1995).

Questions	Aswers	Notes
1. How widely reused are common assets?	Varies	Some (e.g., the Unix tools) are widely reused others (e.g., Cosmic) are not
2. Does programming languages affect reuse?	No	
3. Do CASE tools promote reuse?	No	
4. Do developers prefer to build from scratch or to reuse?	Prefer to reuse	
5. Does perceived economic feasibility?	Yes	
6. Does reuse education influence reuse?	Yes	Corporate training is especially essential
7. Does Software Engineering experience influence reuse?	No	
8. Do recognition rewards increase reuse?	No	
9. Does a common software process promote reuse?	Probably	Respondents say no, but reuse levels suggest belief in the efficacy of a common process helps
10. Do legal problems inhibit reuse?	No	May change in the future
11. Does having a reuse repository improve code reuse?	No	
12. Is reuse more common in certain industries?	Yes	More common in telecommunications, less in aerospace
13. Are company, division, or projects sizes predictive of organizational reuse?	No	
14. Are quality concerns inhibiting reuse?	No	May change in the future
15. Are organizations measuring reuse, quality, and productivity?	Mostly no	
16. Does reuse measurement influence reuse?	No	Measurement probably not being used

Eight years after the work presented by Frakes & Fox, Robert Glass (Glass, 2002) published an important book related to software engineering in general. The book contains 55 Facts (F), which 6 are associated to software reuse:

F1. Reuse-in-the-small (libraries of subroutines) began nearly 50 years ago and is a well-solved problem. According to Glass, although the idea of software reuse had been associated to McIlroy's work, in the mid-1950, some organizations such as IBM applied software reuse in its projects. In IBM, a library of small cientifc software routines (math, sorts, merges) was developed to allow reuse in the projects.

F2. Reuse-in-the-large (components) remains a mostly unsolved problem, even though everyone agrees it is important and desirable. Glass correctly advocates that it is one thing to build useful small software componentes, while it is another to build useful large ones. For him, there are a lot of different opinions about this question ranging from the 'NIH (Not-Invented-Here) syndrome' to specific skills for this task. For Glass and Practioners, the most plausible is software diversity and the difficulty of building one component suitable for different applications.

F3. Reuse-in-the-large works best in families of related systems and thus is domain-dependent. This narrows the potential applicability of reuse-in-the-large. Fact

2 showed that reuse-in-the-large is very dificult to achieve, but one way to make it to work is to develop components that can be reused within a domain. Thus with the uses of techniques such as domain engineering or software product lines a set of assets (requirements, architectures, test cases, source code, etc.) can be reused in an effective way.

F4. There are two "rules of three" in reuse: i. It is three times as difficult to build reusable components as single use components, and ii. a reusable component should be tried out in three different applications before it will be sufficiently general to accept into a reuse library. The first rule is about the effort needed to build reusable components. As we explained, to develop reusable software is a complex task and often, the person responsible for this task should think about generalizing the particular problem to be solved by trying to determine whether there is some more general problem analogous to this specific one. Thus, the reusable component must solve this general problem in such a way that it solves the specific one as well. Moreover, not only must the component itself be generalized, but the testing approach for the component must address the generalized problem. In this way, the complexity of building a component arises in requirements, design, coding and testing.

The second rule is about being sure that the reusable component really is generalized. It is not enough to show that it solves the specific problem, but it must solve also some related problems, problems that may not have been so clearly in mind when the component was being developed. Thus, the idea is that the component can be used in at least three different applications before concluding that it is truly is generalized. According to Glass, the number three is arbitrary and in his case, based on his experience.

F5. Modification of reused code is particularly error-prone. If more than 20 to 25 percent of a component needs to be revised, it is more efficient and effective to rewrite it from scratch. The initial facts discussed the difficult of performing reuse-in-the-large scale, except for families of systems, mainly because of the diversity of the problems solved by software. In this context, some researchers believe that the silver bullet was a careful modification, where components can be used anywhere, even in totally unreleted families of systems.

Glass considers that this idea has problem also, because of the complexity in maintaining existing software, the difficulty of comprehending the existing solution (even for the developer who originally built the solution) and the hard task in mainting the documentation updated.

F6. Design pattern reuse is one solution to the problems inherent in code reuse. The previous facts have been discouraging about software reuse: reuse-in-the-small is a well-solved problem; reuse-in-the-large is a difficult problem to solve, except within families of systems; and modifying reusable components is often difficult and not a trivial idea. In this context, one question is: "What is a developer to do to avoid starting from scratch on each new problem?". One solution that software practioners have always used is to solve their problems remembering yersterday's solution, mainly at the design level. Design reuse had a strong impact in the 1990s, when it was packaged in the form of Design Patterns, a description of a recurrent problem, accompanied by a design solution to that problem.

2.14 Software Reuse around the World

There are a set of initiatives around the world in order to research, develop and put in practice a lot of software reuse techniques. All of these initiatives are looking to gain the benefits of adopted well-defined techniques, methods, models, processes, environments and tools for software reuse. In this section, the most relevant and active software reuse efforts around the world will be presented.

2.14.1 Software Technology for Adaptable Reliable Systems (STARS)

From 1988 up to 1996, Mark Simos with a number of colleagues from companies including Lockheed Martin, Hewlett Packard, the DARPA STARS Program, Change Management Consulting, and Synquiry Technologies, Ltd. developed a corpus of methodology materials and collateral addressing the technical, cognitive and organizational challenges of systematic software reuse, and more generally, knowledge creation within software-intensive organizational settings.

The Software Technology for Adaptable Reliable Systems (STARS)[1] project produced several documents on methodology for software reuse. Many of these

[1] http://domainmodeling.com/stars.html

documents have been cleared for public use, but have not been generally available since 1999. A set of projects were developed with this initiative, such as: Organization Domain Modeling (ODM), and Reuse Library Framework (RLF), Reuse-Oriented Software Evolution (ROSE), among others.

All the projects developed were produced until 1996 and after that, these projects does not contain any information available on the Web site, only scientific reports relating to the usage of this project cited above.

2.14.2 Korea Industrial Technology Foundation (KOTEF)

The Korea Industrial Technology Foundation (KOTEF) is an initiative of the POSTECH Software Engineering Laboratory from Korea. This group started their research in 1990, but the software reuse projects started recently. The KOTEF initiative occurred from 2003 until March 2005, focusing on the development of methods and tools for product line software engineering.

There is a lack of information about techniques, methods, processes and results adopted by this initiative on the web-site[2]. However, this initiative is much cited in some technical papers and computer-science workshops, thus the impact of this initiative on the emergence and competitiveness of the Korea software industry will be very large in the future.

2.14.3 Software Reuse Initiative in USA

From 1994 to 2000, the Advanced Technology Program's (ATP) provided a total of $42 million in public funds to support 24 separate technology development projects in this emerging field, such as CBD, and product-lines, among others (White & Gallaher, 2002).

Over the course of three rounds of funding, ATP committed a total of nearly $70 million and industry committed an additional $55 million of cost-sharing funds to a total of 24 projects. Of the 24 projects, 18 were completed by the time of this study, two are still under way, and four failed to complete. Two-thirds have yielded commercial products. To date, three-quarters of the projects have reached the commercialization phase even though some of the resulting products are not yet generating revenues.

[2] http://selab.postech.ac.kr

According to the ATP, the lessons learned from the failure cases help them to understand better some aspects of software reuse projects and aids them in next selections of projects for support.

2.14.4 Software Reuse Initiatives in Japan

According to Aoyama et al. (Aoyama et al., 2001), around 1995, a set of research and development programs on CBSE were initiated in Japan. However, these initiatives do not contain some information and could be summarized as follows:

- From 1996 to 1998, 21 major computer and software companies, sponsored by the government's Information Technology Promotion Agency (IIPA), started a lot of research in CBSE key-fields;
- In 1998, the Workshop on Component-Based Software Engineering (CBSE) was first held in Kyoto/Japan in conjunction with the International Conference on Software Engineering (ICSE). The main CBSE researchers around the word were present;
- From 1998 to 1999 the ITC (Information Technology Consortium program), another industry consortium of 12 major software houses, conducted a series of pilot developments of applications, infra-structures and components;
- From 1996 to 2000, JIISA (Japan Information Service Industry Association), the largest industry association of software development organizations in Japan, formed a working group to transfer CBSE technology to its 600-odd member-companies;
- From 1998 to 2000, SSR (Strategic Software Research) which is a research consortium involving both academia and industry and focusing on information technology. A working group conducted an intensive study of CBSE and software architecture design;
- For a long time, as shown early, the Japanese software industry has been trying to adopted CBSE in its software development life-cycle through a lot of research and initiatives from the government and industry; and
- In order to aid the software component marketplace's maturation, 30 Japanese COM component vendors formed the ActiveX Vendors Association (AVA) in

1997. Another initiative is from Vector Inc., who list more than 50.000 individual shareware and open-source items for download on their Web site.

As shown, CBSE is a strategic research and development issue in these countries. And, Aoyama believe’s that CBSE will provide the competitive edge that will enable software development organizations to more rapidly produce quality software.

2.14.5 Component Industry Promotion (CIP) from Korea

In the late 1990s, the Korean Ministry of Information and Communication (MIC), was inspired by market research and promising forecasts on CBD to begin a CBD initiative (Kim, 2002). After a through investigation and planning, MIC launched a nationwide Component Industry Promotion (CIP) project in January 1999 to promote the development of CBD technologies and Commercial-Off-The-Shelf (COTS) components. The project has been conducted in four main venues: **(i)** developing CBD core technologies, **(ii)** developing a library of COTS components, **(iii)** promotion and training in CBD, and **(iv)** developing relevant standards. Over 100 software companies have been supported by IITA during the first two years.

Due largely to the CIP project, CBD is rapidly becoming the main software development paradigm in Korea. Software development projects for government are strongly encouraged to utilize CBD technology, and IT companies have already started CBD projects. The CIP project was challenging initially, but now the regional IT industry is successfully integrating and utilizing the advances and standards of the project.

Furthermore from 1999 to 2001, there have been active exchanges of research results and ideas among Asian countries. The Asia-Pacific Software Engineering Conference (APSEC) is the main conference held in the region, focusing on software engineering issues. In conjunction with APSEC 1999, a workshop on component-based development and software architecture was organized. In addition to APSEC, an annual forum on CBD has been held in the region. The first forum on Business Objects and Software Components (BOSC) was held in Seoul in 2000, the second BOSC forum was held in October 2001 in Tokyo, and the next forum will be in China. This is a major conference specializing in CBD technology in the region.

2.14.6 Engineering Software Architectures, Processes and Platforms for Systems-Families (ESAPS)

The Engineering Software Architectures, Processes and Platforms for Systems-Families (ESAPS)[3] started on 1999 and it aims to provide an enhanced system-family approach to enable a major paradigm shift in the existing processes, methods, platforms and tools. ESAPS is carried out by a consortium of leading European companies, research and technology transfer institutions and universities.

The ESAPS consortium is structured around a number of main companies, which act as prime focal points. Each offers a complementary set of engineering capabilities to the project. These companies have a common need for advanced system-family technology. This technology has been researched by these companies in successful European co-operation projects. This experience has to be transferred to industrial practice. In the ESAPS project, the companies act as regional centers of excellence. Several universities and other companies are associated with each prime contractor forming a regional cluster.

Until today, many workshops and definitions of processes, methods and models for software reuse were developed. Some real case studies of these proposals were developed in several European companies, such as: Alcatel, Bosch, Market Maker, Nokia, Philips, and Siemens, among others. However, all the proposals, results and evaluations of this research group were accomplished between 1999 and 2002. After that, no results were published or available on the web-site.

2.14.7 From Concepts to Application in System-Family Engineering (CAFÉ)

From Concepts to Application in System-Family Engineering (CAFÉ) started in 2001 and it aims to provide a path from ESAPS concepts to application. CAFÉ is carried out by a consortium of leading European companies, research and technology transfer institutions and universities.

The ESAPS project (presented earlier) develops concepts for product family engineering, based on its core process. The CAFÉ project intends to bring these

[3] http://www.esi.es/en/Projects/esaps/esaps.html

concepts to maturity so that they can be applied in concrete projects by developing methods and procedures from these concepts. It is based on the same core process, with focus on the very early and late sub-processes. The results of the CAFÉ project, encompassing the structure of the assets and the knowledge about methods and procedures, will be used for tools and concrete applications.

The same sets of companies that support the ESAPS project participate and collaborate with this project too. Furthermore, this project extends the work done in ESAPS and the main results could be obtained since 2001 until 2004. Some documents and workshops are available on its web-site[4] for consulting.

2.14.8 Software Reuse Initiative in Europe

According to McGibbon et al. (McGibbon et al., 2001), there are some countries in Europe, such as United Kingdom, Scandinavia, Germany, and other “northern” regions who have a strong engineering bias, so CBSE has enjoyed an earlier and wider adoption there. Besides this, Germany and England have organized a lot of conferences related to CBSE in order to discuss component-related aspects.

McGibbon et al., based on its experience as a consultant to large and small organizations throughout Europe, explored the successes and failures of software component-based development on a regular basis across a wide variety of businesses and technologies in Europe.

Perceived risks – both in technology and in business culture – remains a major barrier to CBSE for many European organizations. Risk-averse organizations will ensure that their transition to CBSE is as controlled and managed as is practical. This means that the focus is on adoption at the project level rather than the enterprise level. McGibbon et al. observed large organization with a centralized culture that initially attempted enterprise-scale adoption of components. After months of research, consultancy support, confusion and deliberation, there organizations eventually found that they needed to focus at the project level to explore and understand the issues involved with CBSE.

Anther interesting factor is that in Europe, many customers state that they will have confidence only in software components that have been produced under a recognized quality control standards such as ISO 9001, BS 5750 (a collection of UK standards for all aspects of quality-driven manufacturing, testing, and installation) and AQAP (a fading standard used by NATO countries on defense contracts). Many software development organizations claim to follow quality standards but fail to deliver quality parts. In general, today's European Union organizations rely only on their own components.

CBSE is currently restricted to systems integrators, middleware, and tools development in Europe. These development groups usually comprise the early adopters market, as they seek any approach that improves the economics of software development. Moving to components is inevitable for such software development organizations, and others are starting to realize the benefits.

2.14.9 Software Reuse Initiative in Brazil

The main challenge of the Reuse in Software Engineering (RiSE)[5] group, from Brazil, is to develop a robust framework for software reuse (Almeida et al., 2004). This group has started in 2004 and until today many papers and software reuse projects have been developed in conjunction with some Brazilian software factories and universities in order to increase the competitiveness of software organizations.

The framework for software reuse has two layers (Figure 2.1). The **first layer** is formed by best practices related to software reuse. Non-technical factors, such as education, training, incentives, and organizational management are considered. This layer constitutes a fundamental step before of the introduction of the framework in the organizations. The **second layer** is formed by important technical aspects related to software reuse, such as processes (reuse, reengineering, adaptation and component certification), environments, and tools (repository systems and its associated tools). This framework constitutes a solid basis for organizations that are moving towards an effective reuse program. Its elements not only help the organization in adopting reuse, but also guide it in the migration process, reducing its risks and failure possibilities.

[4] http://www.esi.es/en/Projects/Cafe/cafe.html
[5] http://www.rise.com.br

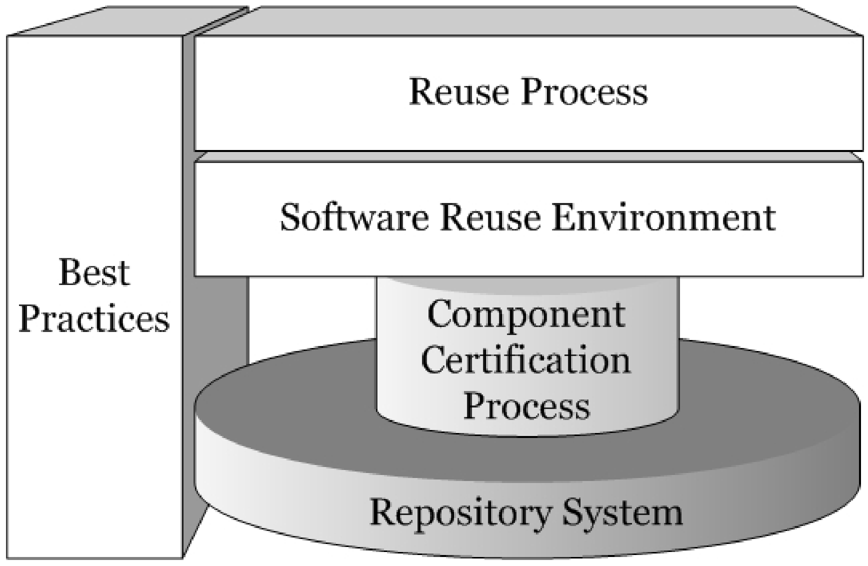


Figure 2.1. A Robust Framework for Software Reuse.

Another important point is that the framework is being developed in conjunction with a large Brazilian software factory (C.E.S.A.R.)[6], where the results could be applied in an industrial environment. This is fundamental for the research in the software reuse area, where small, non-realistic examples often takes place.

2.15 Key Developments in the Field of Software Reuse: Summary

Figure 2.2 summarizes the timeline of research on the software reuse area[7], in which we mark (with "X") the milestones of research in the area. Initially, McIlroy (McIlroy, 1968) proposed to investigate mass-production techniques for software, according to patterns derived from the construction industry. His work represented the initial effort related to software reuse, discussing the development of a component sub industry.

Next, in 1987, Prieto-Diaz & Freeman (Prieto-Diaz & Freeman, 1987) presented a partial solution for the component retrieval problem. Their work proposed a faceted classification scheme based on reusability-related attributes and a selection mechanism as a solution to the problem. This work has influenced strongly the research in the component retrieval area.

[6] http://www.cesar.org.br
[7] Specific aspects related to cost models and software reuse metrics were not considered. As well as: research and evolution of Domain Engineering, Component-Based Development (CBD) and Software Product Lines.

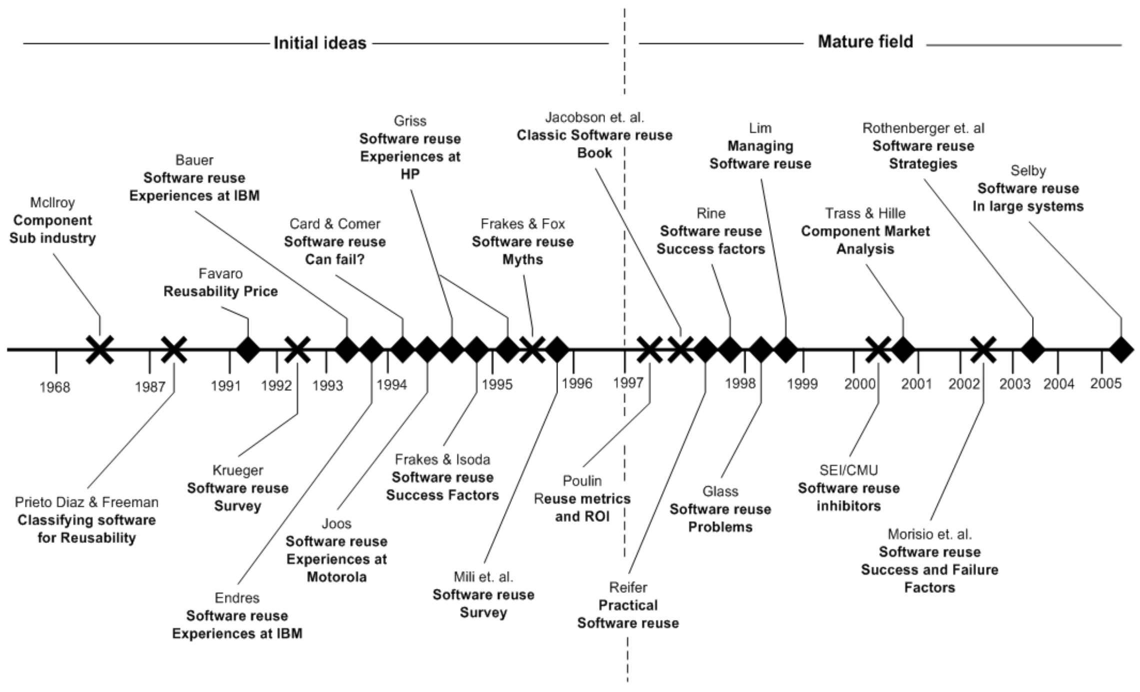

Initial ideas
Mature field
McIlroy Component Sub industry
Bauer Software reuse Experiences at IBM
Favaro Reusability Price
Card & Comer Software reuse Can fail?
Griss Software reuse Experiences at HP
Frakes & Fox Software reuse Myths
Jacobson et. al. Classic Software reuse Book
Rine Software reuse Success factors
Lim Managing Software reuse
Trass & Hille Component Market Analysis
Rothenberger et. al Software reuse Strategies
Selby Software reuse In large systems
1968
1987
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
Prieto Diaz & Freeman Classifying software for Reusability
Krueger Software reuse Survey
Endres Software reuse Experiences at IBM
Joos Software reuse Experiences at Motorola
Frakes & Isoda Software reuse Success Factors
Mili et. al. Software reuse Survey
Poulin Reuse metrics and ROI
Reifer Practical Software reuse
Glass Software reuse Problems
SEI/CMU Software reuse inhibitors
Morisio et. al. Software reuse Success and Failure Factors

Figure 2.2. Research on software reuse timeline.

In 1992, the first important survey on software reuse was presented. Krueger (Krueger, 1992) presented the eight different approaches to software reuse found in the research literature. He uses taxonomy to describe and compare different approaches and make generalizations about the field of software reuse. The taxonomy characterizes each reuse approach in terms of its reusable artifacts and the way these artifacts are abstracted, selected, specialized, and integrated. In 1995, Frakes & Fox (Frakes & Fox, 1995) gave an expressive contribution by analyzing sixteen questions about software reuse using survey data collected from organizations in the U.S and Europe. The findings contributed to breaking some of the myths related to software reuse, as can be seen in Table 2.3.

In the end of the 90’s (1997-1999), the area of software reuse had matured. The sign of maturity can be seen as books (as opposed to workshop position papers, conference and journal papers) start appearing, such as Poulin (Poulin, 1997), Jacobson et al. (Jacobson et al., 1997), Reifer (Reifer, 1997), and Lim (Lim, 1998). Poulin’s book presented important contributions to research related to software reuse, as for example, How to measure the benefit of software reuse? What are the relative costs of developing for and with reuse? In 1997, Jacobson et al. (Jacobson et al., 1997) published the classic book about software reuse. Their book describes how to implement an effective reuse program, addressing various aspects of software reuse, from organizational factors to implementation technologies.

The Software Engineering Institute (SEI) report (Bass et al., 2000) was also important, since it raised several questions regarding component-based software engineering. At the end, in 2002, Morisio et al. (2002) presented and discussed some of the key factors in adopting or running a company-wide software reuse program. The key factors are derived from empirical evidence associated the analysis of 24 projects performed from 1994 to 1997.

2.16 Software Reuse Facilitators

Table 2.4 presents a general summary of facilitators related to software reuse. According to the table, practices such as business strategy, common architecture, design for reuse and the following – highlighted with "yes" – are associated the software reuse success. However, some practices need more research and findings to prove the success, such as formal process, and legal issues. At the end, technological aspects such as the use of repository systems, and the acquisition from new technology do not assure software reuse success, as some companies believe.

Table 2.4. Software Reuse Facilitators.

Practice	**Success**
Business strategy	Yes
Common Architecture	Yes
Components	Yes
Design for Reuse	Yes
Economics	Yes
Education	Yes
Formal Process	Probably
Human Factors	Yes
Incentives	Yes
Legal Issues	Need more research
Planning	Yes
Management	Yes
Measurement	Yes
Reuse Process	Yes
Repository	Does not assure success if used alone
Technology-acquisition	Does not assure success
Training	Yes

This brief analysis shows that software reuse is more related to non-technical aspects. Reuse initiatives that focus only on building a repository, and worse, on first building a repository system, will not achieve their goals. Obtaining high-levels of reuse requires a paradigm change, leading to the creation, management, and use of a software reuse culture.

2.17 Software Reuse: The Future

With the maturity of the area, several researchers have discussed the future directions in the software reuse. In (Kim & Stohr, 1998), Kim & Stohr discuss exhaustively seven crucial aspects related to reuse: definitions, economic issues, processes, technologies, behavioral issues, organizational issues, and, finally, legal and contractual issues. Based on this analysis, they highlight the following directions for research and development: measurements, methodologies (development for and with reuse), tools and non-technical aspects.

In 1999, during a panel (Zand et al., 1999) in the Symposium on Software Reusability (SSR), software reuse specialists such as Victor Basili, Ira Baxter and Martin Griss present several issues related to software reuse adoption on a larger scale. Among the considerations presented were highlighted the following points*: i. education in software reuse area still is a weak point; ii. the necessity of the academia and industry work together; and iii. the necessity of experimental studies to validate new works*.

In the same year, Jeffrey Poulin (1999) briefly looks at the many achievements and positive effects that reuse has on many practical problems faced by industry. Poulin considers that problems such as library, domain analysis, metrics, and organization for reuse are solved. However, this vision does not agree with the research community that intensively researches these problems toward solutions. For Poulin, the recent submissions involving software reuse address many of the same topics that have appeared many times in the past and do not recognize previous works. For him, "*reuse R&D needs to focus on the unsolved problems, such as interoperability, framework design, and component composition techniques*".

In 2005, Frakes & Kang (Frakes & Kang, 2005) present a summary on the software reuse research discussing unsolved problems, based on the Eighth

International Conference on Software Engineering (ICSR), in Madrid, Spain. According to Frakes & Kang, open problems in reuse include*: reuse programs and strategies for organizations, organizational issues, measurements, methodologies, libraries, reliability, safety and scalability.*

2.18 Chapter Summary

Software reuse is the process of using existing software artifacts rather than building them from scratch. This process is crucial for companies interested in improving software development quality and productivity, and in costs reduction. In this chapter, we presented a survey on software reuse, discussing its origins, company reports, success and failure factors, myths, and inhibitors for software reuse, in order to guide companies towards systematic software reuse.

The chapter aimed to also identify essential aspects in the software reuse program (see Table 2.4) that sometimes are not considered, such as business strategy, education, human factors, and training. Table 2.4 also presents other aspects that are not as important as they looked, such as the use of repositories and object-oriented technologies, for example.

Next chapter describes the main concepts about one of the techniques that promote software reuse, the Component-Based Software Development, discussing its principles, risks, challenges, inhibitors and other relevant aspects for research and development.

2.19 References

(Almeida et al., 2004) Almeida, E. S.; Alvaro, A.; Lucrédio, D.; Garcia, V. C.; Meira, S. R. L. **RiSE Project: Towards a Robust Framework for Software Reuse**. In: *IEEE International Conference on Information Reuse and Integration (IRI)*, Las Vegas, Nevada, USA. IEEE Press. 2004.

(Aoyama et al., 2001) Aoyama, M.; Heineman, G. T.; Councill, B. **CBSE in Japan and Asia**, In: *Component-Based Software Engineering: Putting the Pieces Together*, Addison-Wesley, 2001.

(Bass et al., 2000) Bass, L.; Buhman, C.; Dorda, S.; Long, F.; Robert, J.; Seacord, R.; Wallnau, K. **Market Assessment of Component-Based Software**

Engineering, In: *Software Engineering Institute (SEI), Technical Report*, Vol. 01, May, 2000.

(Basili & Rombach, 1991) Basili, V. R.; Rombach, H. D. **Support for Comprehensive Reuse**, In: *Software Engineering Journal*, *Special issue on software process and its support*, Vol. 06, No. 05, April, 1991, pp. 306-316.

(Basili et al., 1996) Basili, V. R.; Briand, L. C.; Melo, W. L. **How reuse influences productivity in object-oriented systems**. In: *Communications of the ACM*, Vol. 39, No. 10, October, 1996, pp. 104-116.

(Bauer, 1993) Bauer, D. **A Reusable Parts Center**, In: *IBM Systems Journal*, Vol. 32, No. 04, September, 1993, pp. 620-624.

(Card & Comer, 1994) Card, D.; Comer, E. **Why Do So Many Reuse Programs Fail?**, In: *IEEE Software*, Vol. 11, No. 05, September/October, 1994, pp. 114-115.

(Cox, 1990) Cox, B. J. **Planning the Software Industrial Revolution**, In: *IEEE Software*, Vol. 07, No. 06, November, 1990, pp. 25-33.

(Endres, 1993) Endres, A. **Lessons Learned in an Industrial Software Lab**, In: *IEEE Software*, Vol. 10, No. 05, September, 1993, pp. 58-61.

(Ezran el al., 2002) Ezran M.; Morisio, M.; Tully, C. **Practical Software Reuse**, *Springer*, 2002.

(Frakes, 1993) Frakes, W.B. **Software Reuse as Industrial Experiment**, *American Programmer*, November, 1993.

(Frakes & Isoda, 1994) Frakes, W.B.; Isoda, S. **Success Factors of Systematic Software Reuse**, In: *IEEE Software*, Vol. 12, No. 01, September, 1994, pp. 15-19.

(Frakes & Fox, 1995) Frakes, W.B.; Fox, C.J. **Sixteen Questions about Software Reuse**, In: *Communications of the ACM*, Vol. 38, No. 06, June, 1995, pp. 75-87.

(Frakes & Kang, 2005) Frakes, W. B.; Kang, K**. Software Reuse Research: Status and Future**, In: *IEEE Transactions on Software Engineering*, Vol. 31, No. 07, July, 2005, pp. 529-536.

(Favaro, 1991) Favaro, J. **What Price Reusability? A Case Study**, In: *Proceedings of the First International Symposium on Environments and Tools for Ada*, California, USA, March, 1991, pp. 115-124.

(Glass, 1998) Glass, R.L. **Reuse: What's Wrong with This Picture?**, In: *IEEE Software,* Vol. 15, No. 02, March/April, 1998, pp. 57-59.

(Glass, 2002) Glass, R. L. **Facts and Fallacies of Software Engineering**, *Addison-Wesley Professional*, 1st edition, 2002.

(Griss, 1994) Griss, M. L. **Software Reuse Experience at Hewlett-Packard**, In: *Proceedings of the 16th IEEE International Conference on Software Engineering (ICSE)*, Sorrento, Italy, May, 1994, pp. 270.

(Griss, 1995) Griss, M. L. **Making Software Reuse Work at Hewlett-Packard**, In: *IEEE Software*, Vol. 12, No. 01, January, 1995, pp. 105-107.

(Jacobson et al., 1997) Jacobson, I.; Griss, M.; Jonsson, P. **Software Reuse: Architecture, Process and Organization for Business Success**, *Addison-Wesley*, 1997, p. 497.

(Jezequel & Meyer, 1997) Jezequel, J. M.; Meyer, B. **Design by Contract: The Lessons of Ariane**, In: *IEEE Computer*, Vol. 30, No. 01, January, 1997, pp. 129-130.

(Joos, 1994) Joos, R. **Software Reuse at Motorola**, In: *IEEE Software*, Vol. 11, No. 05, September, 1994, pp. 42-47.

(Kim & Stohr, 1998) Kim, Y.; Stohr, E. A. **Software Reuse: Survey and Research Directions**, In: *Journal of Management Information Systems*, Vol. 14, No. 04, 1998, pp. 113-147.

(Kim, 2002) Kim, S. D., **Lessons learned from a nationwide CBD promotion project**, In: *Communications of the ACM,* Vol. 45, No. 10, October, 2002, pp. 83-87.

(Krueger, 1992) Krueger, C.W. **Software Reuse**, In: *ACM Computing Surveys,* Vol. 24, No. 02, June, 1992, pp. 131-183.

(Lim, 1998) Lim, W. C. **Managing Software Re-use**, *Prentice Hall PTR*, 1st edition 1998.

(McGibbon et al., 2001) McGibbon, B.; Heineman, G. T.; Councill, B. **Status of CBSE in Europe**, In: *Component-Based Software Engineering: Putting the Pieces Together*, Addison-Wesley, 2001.

(McIlroy, 1968) McIlroy, M. D. **Mass Produced Software Components**, In: *NATO Software Engineering Conference Report*, Garmisch, Germany, October, 1968, pp. 79-85.

(Mili et al., 1995) Mili, H.; Mili, F.; Mili, A. **Reusing Software: Issues and Research Directions**, In: *IEEE Transactions on Software Engineering*, Vol. 21, No. 06, June, 1995, pp. 528-562.

(Mili et al., 1998) Mili, A.; Mili, R.; Mittermeir, R. **A Survey of Software Reuse Libraries**, In: *Annals Software Engineering*, Vol. 05, January, 1998, pp. 349–414.

(Morisio et al., 2002) Morisio, M.; Ezran, M.; Tully, C. **Success and Failure Factors in Software Reuse**, In: IEEE Transactions on Software Engineering, Vol. 28, No. 04, April, 2002, pp. 340-357.

(Moore, 2001) Moore, M. **Software Reuse: Silver Bullet?**, In: *IEEE Software*, Vol. 18, No. 05, September/October, 2001, pp. 86.

(Poulin, 1997) Poulin, J. S. **Measuring Software Reuse**, *Addison-Wesley*, 1997.

(Poulin, 1999) Poulin, J. S. **Reuse: Been There, Done That**, In: *Communications of the ACM*, Vol. 42, No. 05, May, 1999, pp. 98-100.

(Prieto-Diaz & Freeman, 1987) Prieto-Diaz, R.; Freeman, P. **Classifying Software for Reusability**, In: *IEEE Software*, Vol. 04, No. 01, January, 1987, pp. 06-16.

(Reifer, 1997) Reifer, D. J. **Practical Software Reuse**, *Addison-Wesley*, 1997.

(Rine, 1997) Rine, D.C. **Success Factors for software reuse that are applicable across Domains and businesses**, In: *ACM Symposium on Applied Computing*, San Jose, California, USA, ACM Press, March, 1997, pp. 182-186.

(Rothenberger et al., 2003) Rothenberger, M.A.; Dooley, K.J.; Kulkarni, U.R.; Nada, N. **Strategies for Software Reuse: A Principal Component Analysis of Reuse Practices**, In: *IEEE Transactions on Software Engineering*, Vol. 29, No. 09, September, 2003, pp. 825-837.

(Sametinger, 1997) Sametinger, J. **Software Engineering with Reusable Components**, *Springer- Verlag*, 1997, pp.275.

(Selby, 2005) Selby, R. W. **Enabling Reuse-Based Software Development of Large-Scale Systems**, In: *IEEE Transactions on Software Engineering*, Vol. 31, No. 06, June, 2005, pp. 495-510.

(Visser, 1987) Visser, W. **Strategies in Programming Programmable Controllers: A Field Study on a Professional Programmer**, In: *Empirical Studies of Programmers: Second Workshop*, Norwood, USA, 1987, pp. 217-230.

(White & Gallaher, 2002) White, W. J.; Gallaher, M. P. **Benefits and Costs of ATP Investments in Component-Based Software**, *GCR 02-834*, November, 2002.

(Zand et al., 1999) Zand, M.; Basili, V. R.; Baxter, I.; Griss, M. L.; Karlsson, E.; Perry, D. **Reuse R&D: Gap Between Theory and Practice**, In: *Proceedings of the Fifth Symposium on Software Reusability (SSR)*, ACM Press, Los Angeles, CA, USA. May, 1999, pp. 172-177.

Chapter 3

Component-Based Software Development

The expression Component-Based Software Development (CBSD) has become a buzzword within the domain of software engineering domain over the last few years. The software industry and academia tend to accept the idea of building large computer systems from small pieces called components that had already been built before, increasing productivity during the development time as well as the quality of the final product. This idea fits on one of the basic principles of computer science: divide to conquer. In this context, this chapter presents the main concepts of CBSD will be discussed.

3.1 Software Components

3.1.1 Context

It is believed that, when you are using software components, you are adapting your software to be in a reuse context, because components are ideally known as software pieces that can be reused in many software systems. As presented in the previous chapter, the concept of software reuse is not new. Since the first computer systems began to be developed, there has been an idea of reusing parts of code in others systems with the intention of reducing complexity, costs and improving quality, considering that the reused artifacts had been already tested (McIlroy, 1968).

Software reuse may be differently approached in different contexts. For example, reuse can be done using Design Patterns (Gamma et al., 1995) and structuring systems that are supposed to solve similar problems with the same architecture. Other example of reuse is Domain Engineering (DE) (Prieto-Diaz, 1990), whose intention is to identify, build, catalog and spread a set of reusable software components inside a specific application domain. Therefore, the DE establishes mechanisms which facilitate representing and searching for information about a domain, and then the reuse activity can be improved.

Another way of reusing software, and maybe the most popular, is software components. If it is considered that software is composed of a set of parts, then, all software is composed of components, because they came out from the problem decomposition, a standard technique used to solve computing problems *(divide and conquer*) (Bachmann et al., 2000). In this context, components and reuse complement each other, in other words, from the moment that components are being used for building a system then they automatically indicate software reuse.

Nowadays, the large need for maintenance of software systems maintenance has become a critical factor for information technology companies because approximately 50% of the development costs are related to maintenance tasks (Gold & Mohan, 2003). Large systems built in the 70's and 80's, for example, needed to be completely remodeled and rebuilt to attend the same requirements and new ones that emerge during the system lifetime. This scene encourages software engineers and architects to design adaptable systems to facilitate the maintenance, allowing parts or "pieces" to be replaced attending new requirements and new market demands. So, using components may be the most appropriate solution, considering that, instead of replacing all the system during the years, for example, pieces of software or components may be added, removed or replaced, attending to new possible requirements.

According to Sametinger (Sametinger, 1997), "*Reusable, adaptable software components rather than large, monolithic applications are the key assets of successful software companies*". (pp. 05)

3.1.2 Motivation

As it has been said in the last section, adopting CBSD leads directly to software reuse. The idea of constructing large systems from small pieces (components) can reduce the development time and improve the final product quality (Szyperski, 2002), then it is possibly the greatest motivation for using this approach.

The reuse activity is composed of many techniques that have the objective of taking the maximum advantage of previously completed work in different phases such as analysis, design and assembly. The target is not to redo the same things, but organize the work that has already been done and adapt it to a new context. Once the improvements are applied to a piece of the project, all the other projects that reuse that piece are supposed to have the quality increased.

So, motivated by the software reuse idea and Component-based Development (CBD), this chapter explains the main aspects about components.

3.1.3 Definitions

As the concept of software reuse, the idea of components is also not new. Many definitions can be found in the literature (Heineman & Councill, 2001). In 1968, McIlroy (McIlroy, 1968) proposed that components could be largely applied to different machines and different users and should be available in families arranged according to precision, robustness, generality and performance. According to him, components could be used to maintain the software industry mass production.

In around 30 years later, Sametinger (Sametinger, 1997) defined: "*Reusable software components are self-contained, clearly identifiable artifacts that describe and/or perform specific functions and have clear interfaces, appropriate documentation and a defined reuse status*." (pp. 68)

Another definition well accepted in the academy is Heineman's (Heineman & Councill, 2001): "*A software component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a component standard*" (pp. 07). For him, "*A component model defines specific interaction and composition standards. A component model implementation is the dedicated set of executable software elements required to support the execution of components that conform to the model*" (pp. 07). He also defines the software component infrastructure as "*a set of interacting software components designed to ensure that a software system or subsystem constructed using those components and interfaces will satisfy clearly defined performance specifications*" (pp. 07).

Besides Heineman's, one satisfactory definition, another is presented by Szyperski (Szyperski, 2002): "*A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties*." (pp. 41)

In this chapter, only the more recent definitions of component from Heineman and Szyperski will be explained with more details in the next sections.

3.1.3.1 Heineman's detailed definition

It is important to make clear all of the elements of Heineman's definition. The following terms used in the definition must be clearly described:

- **Standard –** an object or quality or measure serving as a basis to which others should conform, or by which the accuracy quality of others is judged;
- **Software element –** a sequence of abstract program statements that describe computations to be performed by a machine;
- **Interface –** an abstraction of the component behavior that consists of a subset of the interactions of that component together with a set of constraints describing when they may occur;
- **Interaction –** an action between two or more software elements; and
- **Composition –** the combination of two or more software components yielding a new component behavior at a different level of abstraction.

As these terms have been explained, the complete Heineman's definition must be revisited. He says that: "*A software component is a software element that conforms to a* ***component model*** *and…*". A component model operates on two levels: first, a component model defines how to construct an individual component; second, the model enables composition by defining an interaction standard that promotes unambiguously specified interfaces. Section 3.2 treats Component Models in more detail.

Another part of the definition says that: "A component model defines specific **interaction and composition standards**. A **component model implementation** is the dedicated set of executable software elements…". For Heineman and also for Szyperski, a component may have an explicit context dependency, and then an interaction standard helps to identify which type of these dependencies it may have. A composition standard defines how components can be composed (using interfaces) to create a larger structure and how a producer can substitute one component to replace another that already exists within the structure.

Complementing the definition, Heineman defines a component model implementation as a "dedicated set of executable software elements necessary to support the execution of the components within a component model". (pp. 12)

3.1.3.2 Szyperski's detailed definition

According to Szyperski, the terms **component** and **object** are used with the same meaning many times. So, a differentiation has to be made essentially to avoid future problems. For him, a component:

i. **Is a unit of independent deployment** – this implies that the component may be deployed separated from its environment and from others components. It can be assumed implicitly that a component can not be partially deployed;

ii. **Is a unit of third-party composition** – for a component to be an element of a composition with a third party, it needs to be self-contained. Also, it needs to have an unambiguous documentation that specifies clearly what it requires and provides. In other words, a component needs to encapsulate its implementation and interact with its environment by means of well defined interfaces; and

iii. **Has no (externally) observable state** – a component can not be distinguished from copies of itself, it means that a component can not be instantiated as objects are**.**

The concepts of instantiation, identity and encapsulation lead to the concept of objects. Comparing to the component definition already cited, an object:

i. **Is a unit of instantiation which has its unique identity** – this identifier points to only one object without considering changes in its internal state during the entire object lifecycle. For example, a car with new tires is still a car;

ii. **May have state and this can be externally observable** – an object allows the observation of its internal state from its class; and

iii. **Encapsulates its state and behavior** – the object class also allows the encapsulation of its state and behavior.

The Szyperski's definition says that a "*A software component is a unit of composition with* ***contractually specified interfaces****…*". In this context, interfaces are the access points for the services provided by the component. Naturally, a component can have multiple interfaces corresponding to multiple access points. Each access point can be accessed by different clients according to their needs. It is important to emphasize the contractual aspect of the interface concept, once the clients and components are developed completely separated and because of the established contract (interface) the interaction between them works fine.

The concept introduced by Szyperski has references to which environment the components are inserted. In other words, a component must specify the necessary deployment environment to provide its correct function. This necessity is called *explicit context dependencies*, referring to the context of composition with another components and deployment. The component model defines rules for composition and the component platform defines rules for deployment, installation and activation.

3.1.4 Attributes

For better classifying components, it is needed to clarify some attributes related to its definition. According to Sametinger (Sametinger, 1997), six attributes are essential for CBD: *Functionality*, *Interactivity*, *Concurrency*, *Distribution*, *Forms of Adaptation*, and *Quality Control*. The following sections treat each cited attribute.

3.1.4.1 Functionality

The component functionality is essential for its reuse in some contexts. Depending on the quantity of operations executed by the component, it may be too specific for a context or too general, or even incomplete. The concept of functionality is complemented with the concepts of applicability, generality and completeness.

i. The *applicability* of a component indicates its likelihood to be a reuse candidate in the range of software systems for which it was designed to be reused. The component's applicability can be high, for a certain application domain, and low or zero for others. For example, a component built to control a car brake system has high applicability in the automobile domain but has zero applicability within the mobile phones domain;

ii. The *generality* indicates if a component has more specific functionality or not. For example, a component that sorts numbers has less generality than a component which sorts arbitrary objects. High generality also means high applicability. However, care has to be taken, because excessive generality leads to more complex components and they can consume time and resources unnecessarily; and

iii. The *completeness* of a component describes if it offers the expected functionality in the applied domain. One classical example of an incomplete component is a menu screen which needs to be painted on a device, like a mobile phone, and does not implement the *paint* method to show its content.

3.1.4.2 Interactivity

Interactive components receive from other environments, which can be other components or subsystems, different inputs in many different situations. For better understanding the concept of interactivity, we can establish a parallel between functions and objects, demonstrating how components may or may not interact.

Functions transform a system from an initial state to a final state through some computation. They do not have memory, in other words, they do not keep the internal state and, for the same input, they will always give the same output. However, objects react to messages received from other objects and resend other messages to the senders performing some computation. Despite functions, an object can keep its internal state and can react in many different ways to the same input according to its state.

An attribute which can influence the reuse of a component is its interactivity. A component may interact with others or with the user. The main target to be reached when the subject is interactive components is: high cohesion and low coupling. This means that components should have a high degree of conceptual unit and that the dependency on other components should be small.

High coupling can discourage the reuse of a component even if it is technically possible, because all the other components on which it depends might have to be incorporated into the design.

3.1.4.3 Concurrency

The parallel execution of components is called concurrency. A concurrent component become non-deterministic, in other words, may give different results for the same input. It happens because a component does not know exactly which sequence of instructions is being executed at any given moment. In such way, this attribute is used to solve non-deterministic problems which do not need sequential execution. The concurrency is also used to gain execution speed and to eliminate potential processor idle time.

3.1.4.4 Distribution

Distributed systems, by definition, are physically or logically separated systems. (Tanenbaum, 2002). The main reasons for the popularity of distribution are not cost considerations but increased capabilities, greater flexibility of incremental expansion and choice of vendors. It is possible to use, buy or sell components for a variety of platforms (Sametinger, 1997). A component that has this attribute of distribution should support being inserted into a distributed environment.

A distributed system consists of independently executing components. They can be implemented in the same programming language using communication mechanisms provided by that language or can also be implemented in different programming languages and paradigms only making use of language-independent mechanisms of communication. Besides data sharing, message passing is the central means for these components to communicate.

3.1.4.5 Forms of Adaptation

Before a component is going to be inserted into a system, some adaptations are required on it. This phase is called **optimization** of a component, on which it will pass through some adaptations that do not affect its essential behavior. After that, a component may pass through a **modification**, what is not recommended because, in most of the cases, it decreases the reusability considering that the primary functionalities may be changed.

A simple example of optimization can be imagined if a graphical menu component is going to be displayed on a mobile phone screen, which may have different dimensions. Then the component would be optimized to work fine in that specific screen size.

Another example for modification would be the case of source code copied from one to another application and then changed to be compatible with the new application. This action may throw negative points, such as different versions of the component running, could bring some inconsistencies into the source code.

3.1.4.6 Quality Control

Quality assurance for components is an extremely complex task. On the other side, verifying this quality is even harder. The process is so complex that there is no formal verification of quality even for small components. The larger of them, written in many different programming languages and for different operating systems, does not make possible component verification with the intention of applying to it a quality stamp. Some works on this area has been investigating it and presenting meaningful result, like the work proposed by Alvaro (Alvaro et al., 2005a) (Alvaro et al., 2005b) (Alvaro et al., 2005c). The main objective of this work, is to define a software component certification process. However, in the actual methods and process, the idea is to develop a fault tolerant system, even if it is composed of unreliable components.

Some tricks are used for ensuring that a component is also fault tolerant, such as pre and post conditions, constants and exceptions (Sametinger, 1997):

i. **Pre-conditions** are Boolean expressions that check if the given input parameters are valid and if the object is in a reasonable state to execute the requested task;

ii. **Post-conditions** check if, after finishing the requested task, it was finished successfully according to the contract that the method or function is supposed to execute;

iii. **Constants** are used every time that a method passes the control to a separate object or component; and

iv. **Exceptions** represent another kind of quality assurance for the component. They are used to handle atypical situations that happen during code execution. An exception can be thrown in the moment that the system requires to read a file that is corrupted, for example.

3.1.5 Composition

According to Heineman (Heineman & Councill, 2001): "*Component composition or assembly is the combination of two or more software components that yields a new component behavior*" (pp. 08). The standards used to make component composition allow the creation of large structures by connecting them or inserting and substituting then within an existing structure. Sometimes, this structure is called framework. A framework can also be defined as a support structure in which another software project can be organized and developed. It may include support programs, code libraries, a scripting language, or other software to help develop and *glue together* the different components of a software project.

Components in a framework can interact with each other, typically through method invocation. For Heineman, there are two basic types of component interactions: *client/server* and *publish/subscribe*. In the first type, components may act as clients requesting information using method invocation. In the second, it may register itself with another component and receive notifications of a predefined event. Also, the component model must support this kind of interaction through the defined interfaces.

There are many different approaches to component composition at different levels (Weinreich, 1997). Components can be composed using all-purpose programming languages, scripts, glue languages, visual programming or composition tools. Glue languages, such as Visual Basic or JavaScript, for example, support composition at a higher level than all-purpose languages, such as C++ and Java. The big problem of using composition language is that glue code must be written or graphically specified from the beginning. Maximum reuse is reached when the component infrastructures are designed for a specific domain where the interactions between the components are predefined.

Component infrastructures enable not only the reuse of individual components but of an entire design. A clear example of that is described by Weinreich (Weinreich, 1997) with a trader-based component infrastructure for graphic editors. According to Heineman, only well-designed component infrastructure enables the effective and efficient assembly of components.

3.1.6 Taxonomies

After understanding the main concepts of software components, one key point to be studied is about the components taxonomy. It may facilitate on component classification and recovery and also makes the certification process possible. This process evaluates components maturity ensuring the final product quality when applying CBD.

In this Section, two different taxonomies will be presented: Heineman's (Heineman & Councill, 2001) and Sametinger's (Sametinger, 1997).

3.1.6.1 Heineman Taxonomy

According to Heineman **(**Heineman & Councill, 2001), components are not created identically. They differ in complexity, scope, functionality level, set of abilities needed to use them and required infra-structure. To facilitate differentiation, Heineman divides components in three categories, as shown in Figure 3.1.

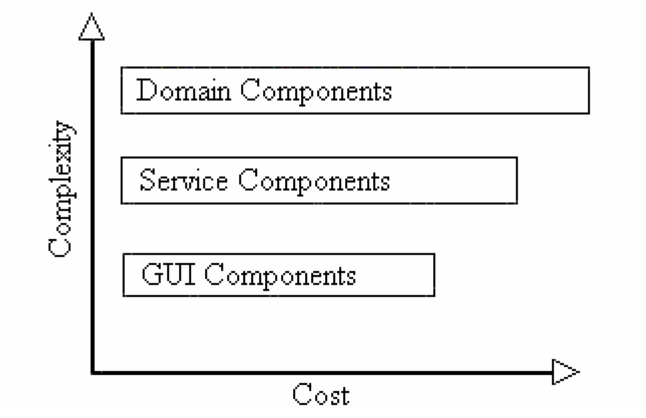


Figure 3.1. Heineman Taxonomy (Heineman & Councill, 2001).

This taxonomy differentiates the components by cost and complexity. Each type of component is explained in details, from the lowest to the highest complexity level (Heineman & Councill, 2001):

- **GUI Components**: they are the most common components found in the marketplace because they have a low complexity level. Simple examples are: buttons, sliders, text fields and other widgets used in building user interfaces for applications. Reusing such pre built components is fairly easy and has a quick payback. Reusing components from this category may increase the productivity in about 40%;

- **Service Components**: this category introduces more complexity to the components. They provide necessary common services for the applications. Also, they include database access, access to messaging and transaction services and system integration services. One common characteristic of service components is that all of them use additional infrastructure or systems to perform their functions. The deployment cost of this kind of component is high because they may integrate many different systems. Productivity increase, by the reuse of these components may reach 150%, however this value may not take into consideration the high deployment cost; and

- **Domain Components**: they are the most difficult kind of component for developing and reusing. Heineman considers domain components only those which are truly reusable, because the non-reusable domain components have an impact on an organization similar to service components and the same business case can be used to justify them. For example, an application used by an insurance company may include Bill, Policy and Claim components. It is, unlikely, that they would find broad reuse outside that domain. If a class diagram of the application is built, it would be possible to see that these kinds of components are the key abstractions of the application. They are normally monolithic, because they interact with other abstractions, as in Order and Bill system, for example. The use of domain components is more costly because they need a huge infrastructure for being deployed, but, productivity may increase 1000%.

3.1.6.2 Sametinger Taxonomy

Before understanding how components taxonomy is structured according to Sametinger (Sametinger, 1997) ideas, it is fundamental to comprehend the composition concept, which is the process of building applications using components and connecting them through their interfaces. Some types of composition are:

- **Textual**: ASCII text is the most common data format and the most common denominator for many software systems. So, systems with textual input and output have proven successful for automation. Reusing components in series, one component inputting the previous component output, has provided the basis for the textual composition;

- **Functional**: can be considered the most widespread mechanism of component composition. It is based on parameterized functions. The parameters are passed in the moment of function call;
- **Modular** (or packages): happens when different system functionalities are divided into two distinct modules. Each module may be inside a package, following the interconnection concept. In other words, a module or package can aggregate many similar components together;
- **Object-Oriented**: based on object oriented concepts, as inheritance, dynamic biding and polymorphism. A component can extend another and not lose its compatibility or can activate many other components through only one method call due to polymorphism;
- **Subsystem**: smalls systems that, interconnected, produce a larger system. A subsystem may be deployed isolated without dependencies, in theory. It can have many modules and consequently many components;
- **Object Models**: to allow that the component composition works between different platforms, an object model may be used, which means, a better component organization. A practical example for this kind of composition is the Common Object Request Broker (CORBA) (OMG, 2001). The object request broker is responsible for the communication between components;
- **Specific Platform**: the component composition or integration only works under a specific platform, as, for example, only under Windows; and
- **Open Platform**: the component composition or integration works under any platform.

The Figure 3.2 shows a possible taxonomy applicable to software components according to Sametinger.

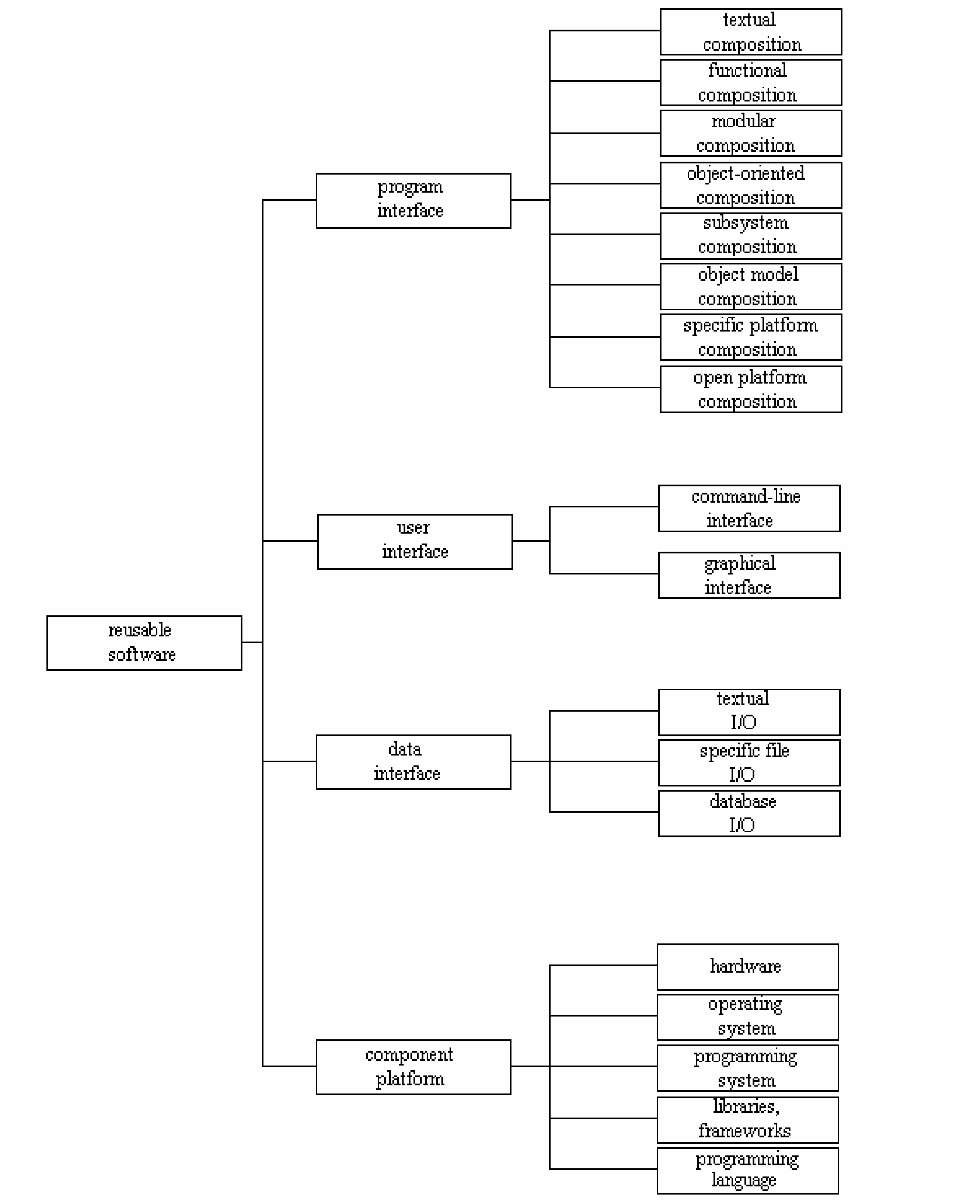


Figure 3.2. Sametinger Taxonomy (Sametinger, 1997).

3.2 Component Models

Component models provide standards for component implementation and interoperability. Also, they provide services and infrastructure such as a meta-information facility, naming and trading services, and transaction monitors. The implementation standards define how components' external interfaces are accessed. These interfaces are the only way to access its functionality externally.

A component model is an indispensable element for the CBSD technology. However strict models can also be a limiting factor in this context (Oberleitner & Gschwind, 2003). In an ideal setting, a developer would be able to use the best components available without having to think about the component model they have been implemented for. Oberleitner & Gschwind proposed a solution to solve this composition problem. The solution is a Java based class framework that allows components access across different models and the construction of new components without worrying about target platforms. Two different factors limit the reuse of components' source code:

i. The standards that define how a component has to be constructed; and

ii. The dependency of the component on services provided only by a certain component model.

The framework abstracts both of these factors using interfaces, which build a meta-model for component models. "A particular component model is supported by writing a plug-in that queries the component model for its component's meta-information, builds a representation of the component and all of its features and provides the required functionality to access these features." (pp. 26) (Oberleitner & Gschwind, 2003)

Other component models can be found in academia like: the KOALA component model (Ommering et al., 2000) which can be used to handle the diversity of software in consumer electronics, specially embedded systems; the CORBA Component Model (CCM) (OMG, 2002); and also the Fractal Project (Fractal, 2005).

This section will talk about the four most commonly known component models: **COM+**, **CORBA**, **Enterprise Java Beans** and **Java Beans**.

3.2.1 COM+

Microsoft's Component Object Model (COM) (Microsoft, 1995) is used heavily within Microsoft's operating systems. COM components are declared using Microsoft Interface Definition Language (MIDL) that supports the description of COM component classes and interfaces. Properties are declared using setter and getter methods having special attributes attached to them. MIDL uses its own type system that is based on the C type system including pointers to interfaces. Components are usually implemented with C++ classes or with another COM-compatible language. Recent additions to COM have been server-side facilities for load-balancing and transaction monitoring, better known as COM+. However, the client side programming model has remained the same. COM+ usage can easily be found on the .NET platform and the proprietary Microsoft languages, for example, Visual Basic .NET and C# .NET.

3.2.2 CORBA

The Common Object Request Broker Architecture (CORBA) (OMG, 2001) has been defined by the OMG to provide an object infrastructure for interoperability among different hardware and software products. A CORBA object is declared by writing an IDL file that contains the object interface definition. This interface takes the definitions of an object's operations and its attributes. The IDL file is compiled by an IDL compiler that generates client stubs and server skeletons for a given language.

The IDL has its own type system which is loosely based on C++. This type system is mapped onto the type system of a given programming language as defined by the corresponding language binding. In the case of the Java programming language primitive types are converted to Java types and CORBA object types are converted to client stubs.

CORBA's communication model is based on object invocation which may be locally or remotely. Each request to a CORBA object is processed by the client stub which forwards the request to an Object Request Broker (ORB) that is located on the host of the client. This ORB uses a communication channel to interact with the ORB on the host of the object's server process. This second ORB forwards the invocation to an appropriate method in the server process.

3.2.3 Java Beans

JavaBeans (Sun Microsystems, 1997) is a simple component model that relies on the Java programming language. Unlike the other component models presented so far, it only supports components executed locally within the client's virtual machine. A

JavaBean is a Java class that has a default constructor and supports the Java serialization mechanism.

JavaBeans support methods, properties and events. Public methods are defined to access the private attributes or properties of the JavaBean (getters and setters). The name of the property is deduced from the method's name. Since a JavaBean is just a Java class, the component model uses Java's type system. An instance of a component is a normal Java object and hence clients access these instances like any other Java object.

3.2.4 Enterprise Java Beans

The Enterprise JavaBeans (EJB) (Sun Microsystems, 2001) component model is an essential part of Sun's J2EE environment and uses the Java type system. EJBs are components that reside within a container on an application server. The implementation of an EJB consists of Java classes that are deployed in the container. Clients use an enterprise bean's home and remote interface to invoke its methods.

Enterprise JavaBeans can implement different concepts. Entity beans model business concepts that are represented in database tables. Session beans model a workflow and thus implement a particular task (Monson-Haefel, 1999). Usually, they are stateless and have no properties. Message-driven beans are similar to session beans but work in message-oriented middleware settings.

Unlike other component models, EJBs do not support events, and also there is no standardized means to find out at runtime if a component uses a particular service provided by the EJB application server.

3.3 Risks and Challenges

Software development suffers chronically from cost overruns, project delays, and unmet user needs. These difficulties can be alleviated through software risk management. It can be defined as "*an attempt to formalize risk oriented correlates of success into a readily applicable set of principles and practices*" (pp. 98) (Ropponen & Lyytinen, 2000). So, how can risk management be done as an attempt to be applied in a component-based software environment?

According to Vitharana (Vitharana, 2003), CBSD encompasses three primary types of stakeholders: *Component Developers*, *Application Assemblers* and *Customers*. Defining the stakeholders eases the risk management into a CBSD environment.

Developers generally construct components under two scenarios:

i. Survey the overall software industry and build components for the mass market; and

ii. Develop components for a specific client or assembler.

Considering a mass-market strategy, developers must identify business areas or domains that would generate enough demand to justify component development. However, building a set of components encompassing the entire domain knowledge and its related business processes is a challenge. As domains are regularly changing them over time, developers must adapt to the changes once the components are produced. Then, after investing considerable resources, developers risk having their component repositories become obsolete due to poor planning or unfavorable industry trends.

Some strategies may be used to workaround the component development problems cited. Developers must also determine the optimal way to fragment the domain into a cohesive set of components. They must also assess how often to release components and how to inform clients, or assemblers, of new versions with bug fixes and enhancements. Other point to be considered is that developing quality components requires the use of a comprehensive testing program. And finally, before entering in component-development projects, a developer must conduct cost-benefit analyses to determine whether to accept a client, or assembler, and contract or construct components for the mass market (Vitharana, 2003).

Assemblers risks and challenges primarily concern the assembly of components in applications, the management of component-based application assembly projects, and the uncertainties of the component market. Crucial challenges when considering this kind of risk include:

- **Matching system requirement specifications**. They are typically documented in formal languages, such as VDM and Z (Harry, 1997), with components in the repository typically coded in some proprietary classification scheme;

- **Demarcating the requirements document into smaller subsets**. Based on the availability of components in the open market, demarcating the requirements into smaller subsets is a tedious process involving many iterations; and
- **Confirming the overall selected component set**. For this point, the most appropriate question should be: does the component satisfy all the requirements of the planned application system?

Probably, the largest problem of using components is the thought about components being unit-tested and bug-proof. However, it does not correspond to the reality. To solve this problem, assemblers might be forced to conduct additional unit testing within the environment in which the component is being deployed. Also, component-based application assembly requires personnel with a different set of skills from those employed in the traditional life-cycle methodologies. Analysts need additional expertise in matching solicited user requirements with components available in the repository before assembling them into applications. These alternatives may reduce the risks of composing a new application from pre-built components.

Customers face both risks and challenges when using component-based applications to meet their enterprise requirements. Problems may happen in managing their component-based and legacy application systems and in achieving and sustaining strategic competitive advantage over their rivals.

One key risk occurs when a system is not capable of satisfying customer requirements. Developers might build systems that do not satisfy customer requirements, due to the market’s lack of suitable components. Customers also face additional risks, such as application quality based on component quality hindering their ability to carry out key business activities. Moreover, there is the bigger problem of satisfying customers’ requirements when they use a set of legacy systems and then these systems need to interoperate with the new component-based software.

Costumers have the ability of influencing the component market, and, with that, they could persuade developers to incorporate their business into the components being built.

Vitharana believes that the risks and challenges propagate from one stakeholder to another. He says: “As assemblers increasingly depend on components developed by developers and customers depend on applications assembled by assemblers, these risks and challenges propagate from one stakeholder to another." (pp. 71). Figure 3.3 helps to understand these cascading effects.

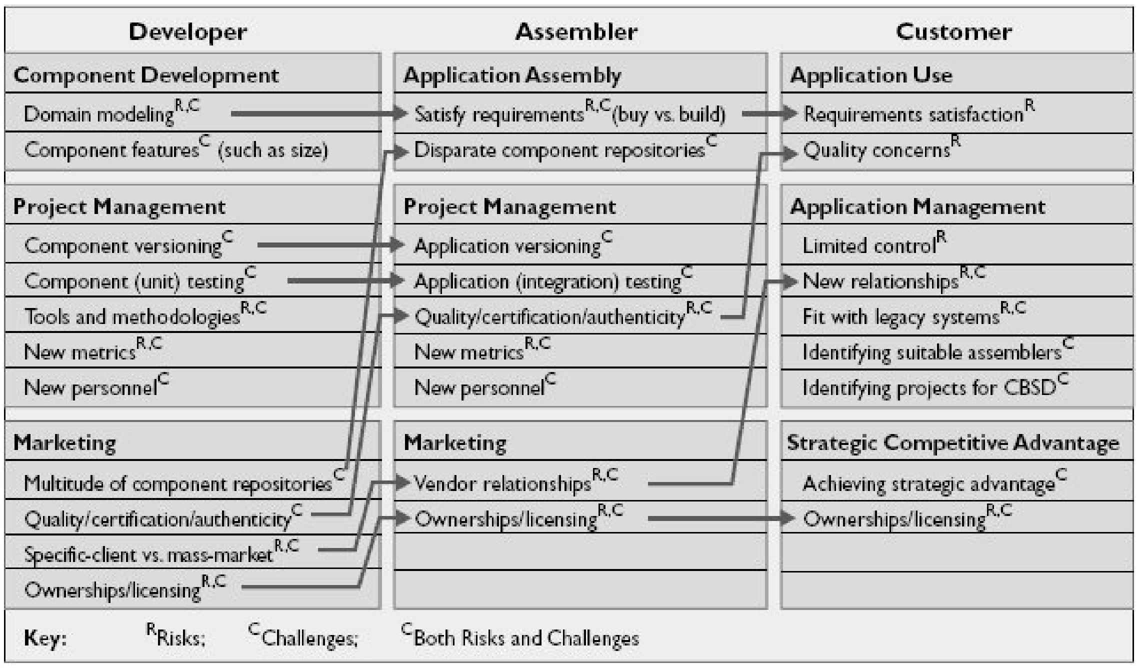

Developer
Assembler
Customer
Component Development
Domain modeling R,C
Component features C (such as size)
Application Assembly
Satisfy requirements R,C (buy vs. build)
Disparate component repositories C
Application Use
Requirements satisfaction R
Quality concerns R
Project Management
Component versioning C
Component (unit) testing C
Tools and methodologies R,C
New metrics R,C
New personnel C
Project Management
Application versioning C
Application (integration) testing C
Quality/certification/authenticity R,C
New metrics R,C
New personnel C
Application Management
Limited control R
New relationships R,C
Fit with legacy systems R,C
Identifying suitable assemblers C
Identifying projects for CBSD C
Marketing
Multitude of component repositories C
Quality/certification/authenticity C
Specific-client vs. mass-market R,C
Ownerships/licensing R,C
Marketing
Vendor relationships R,C
Ownerships/licensing R,C
Strategic Competitive Advantage
Achieving strategic advantage C
Ownerships/licensing R,C
Key: R Risks; C Challenges; C Both Risks and Challenges

Figure 3.3. Key CBSD stakeholder risks and challenges and their cascading effects (Vitharana, 2003).

3.4 Component Markets

The idea of developing software systems from prefabricated software components has a long tradition in software engineering field, component markets have also been part of this idea. The concept of component-based software systems is independent of any programming paradigm. However, it was especially expected to gain a higher degree of software reuse from Object Orientated Programming (OOP) techniques.

Organizations may get benefits from organizing an internal component market. Software components would be reused over multiple projects saving valuable resources. However, the most appropriate marketplace to buy and sell components should be the Internet: an international, freely accessible network, which is perfectly suited for offering, promoting and distributing components (Traas & Hillegersberg, 2000).

It has been predicted huge growth for the software components market. Ovum has estimated the size of component market to be $64 billion in 2002 (Ovum, 2005).

However, the reality is far from that. So, a question could be: does a component market already exist, and if so, what is the current status?

Traas & Hillegersberg did some research into the status of the component market in 2000. They realized that the websites which could possibly be a component market are classified into one of the following categories:

- **Producer:** a website of an organization that sells its own self-constructed components**;**
- **Catalogue:** a website of an organization that compiles a list of hyperlinks to components, based at producers' sites; catalogues do not actually sell components; and
- **Intermediary:** a website of an organization that sells components, built by third parties**.**

The research result demonstrates that the component market is still in infancy and some points can be considered to allow its growth:

i. The market has to focus on fine-grained components. Fine-grained components can be expected to be cheaper and easier to understand than large-grained components. Then it reduces the risk of large transactions to fail and does not allow the companies to spend much money on components that do not corresponds to what was expected;

ii. Suppliers must facilitate for their customers to assess the components' assets or values, by offering extensive information. Then the component certification could be the best alternative;

iii. Components have to be offered against known/published prices (price lists), so the components prices could be compared in many suppliers;

iv. The market has to focus on black-box reuse (Figure 3.4). The intellectual property rights of component suppliers are better protected when source code is not delivered with the component. Also, black box reuse assures that the true principles of CBD are followed, such as adapting the internals of the component is prohibited, reducing the customization cost close to zero. Furthermore, using internally developed components may reduce even more the acquisition costs of the component; and

v. There has to be a (supplier) independent Internet search engine for components, such as Google Code (Google Code, 2005) and Codase - Source Code Search Engine (Codase, 2005).

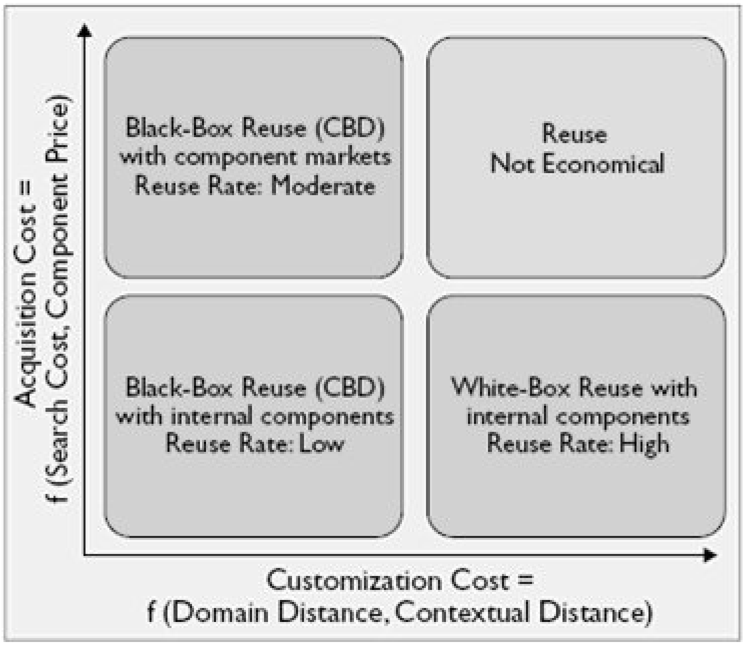


Figure 3.4. Software Reuse Strategies (Ravichandran & Rothenberger, 2003).

3.5 Inhibitors

As shown in the last section, component markets are a correlated idea with software components. However, such markets can be found only rarely in practice, because markets for software components still need to evolve (Turowski, 2003).

The aim of this section is to discuss about drivers and inhibitors of the development of a software component industry, which are considered an important prerequisite for software component markets according to Turowski (Turowski, 2003). We have introduced the concept of modular systems or modular design, that is underlying component-based software systems. Modular systems are the result of a functional decomposition which the basic idea of a component independently deployable (Szyperski, 2002).

For Szyperski four factors can be considered as the main inhibitors for modularity design, as follows:

- **Domain complexity**: this factor contributes by giving an idea of how problematic the component deployment will be within the specified domain. The factor ‘Domain complexity’ has two dimensions:

  i. The cohesion between the elements; and

  ii. The instability of the basic concepts that are underlying the domain.

  Then, these factors may be considered when using modular decomposition, thus they affect directly the system architecture and the components complexity;

- **Customer influence**: according to Turowski, “*we do not assume that the customer has an interest in a modular solution if there is no gain from such architecture. However, we expect that the customer plays an active and necessary role as change agent if it is in his or her interest*” (pp. 06). From this idea, the conclusion for this topic is that customer influence is a direct inhibitor factor to the modular composition, and this influence can possibly not allow the solution to be implemented;

- **Pressure towards a modular design**: it is expected that a modular architecture is important if new functionality has to be often added to the new system. For example, Enterprise Resource Planning (ERP) systems were firstly designed to support scheduling process, and today these systems have a wide range of functionality, like Human Resource Management or Customer Relationship Management. A highly integrated architecture, which is the characteristic for most of the current ERP systems, makes the integration of new functionality or new releases more expensive and complicated. So, it is important to understand that pressure towards a modular design is an inhibiting factor of modular design; and

- **Standards**: are considered the main reference for the activities within a company. In this context, standards are intended to decouple the activities of different agents to enhance the labor division in an industry. The standards inside an organization can decide whether a component will be used or not, by measuring if the information provided is sufficient. So, it allows discriminating between components that are appropriate or not for the purpose.

3.6 Component-Based Development Methods

Developing components requires a systematic approach the same as any other software development process. Therefore, using methods or processes for this kind of development eases the construction of more complex systems that connect all the “parts”. In this scenario, it can be justified that the majority of research, and its results are new CBD methods proposals in order to supply a lot of lacks in this area (Almeida, 2003) (Stojanovic et al., 2001). Among a set of methods, such as *PECOS* (PECOS, 2005), *RUP* (Kruchten, 1999), *Select Perspective* (Stojanovic et al., 2001), two of them can draw more attention to academia: *Catalysis* (D'Souza & Wills, 2001) and *UML Components* (Cheeseman & Daniels, 2001). The following sections describe each one.

3.6.1 Catalysis

Catalysis is a research initiative developed at the University of Brighton by D’Souza and Wills (D'Souza & Wills, 2001). This research had, as result, an integrated method to model and construct object oriented distributed systems. According to them, Catalysis covers all the component development phases, from its specification to its implementation.

This method is based on a set of principles for software development that are: **abstraction**, **precision**, **refinement**, **pluggable parts** and **reuse laws**.

The **abstraction** principle guides the developer to search for the essentials aspects of the system, not getting close to the details that are not relevant for the context. D’Souza and Wills say that “*The only way we can handle complexity is by avoiding it.*” The **precision** principle aims to discover errors and modeling inconsistencies. Successive **refinement** between the phase transitions helps on getting more precise artifacts that are tended to be reused. The **pluggable parts** principle supports the components reuse with the intention of building large systems just plugging the smaller reused parts. Lastly, the main Catalysis **reuse law** is: “*Do not reuse code without reusing the specification of that code.*”

The software development process of Catalysis is divided in three levels:

i. **Problem domain**: specifying “what” the system does to solve the problem;

ii. **Component specification**: where the system behavior is described; and

iii. **Component internal design**: where the internal implementation details are specified.

3.6.2 UML Components

UML Components (Cheeseman & Daniels, 2001), an UML extension, represents a process for component-based software specification. In this process, the systems are structured as four distinct layers: user interface, user communication, system services and business services. This structure represents the system architecture. Besides that, UML Components support the use of UML for modeling all the phases of the development of component-based systems, including activities as requirements definition, identification and description of components’ interfaces, modeling and specification, and also implementation and composition.

The method uses a simple way of extending UML that is to use stereotypes on the entities. It means that the UML purpose, already well disseminated in the modeling area, can be followed by the software analyst. A concern that the authors have with the method is to offer a solution without specifying the platform, so the method could be used in many platforms with many different technologies.

3.7 Components: Current State

After discussing about some component concepts, showing from the basic concepts to the CBSD methods, passing through the component models, markets and inhibitors, it is possible to conclude that software components are the big promise of software industry in order to promote software reuse. It is believed that facing some problems with object oriented paradigm motivates a shift towards component-based software development. However, CBSD has problems that are not fully addressed (or not addressed at all). Some technical aspects of components still have a lack of information and reliability, such as: specification of components, component composition, verification of compositions, quality of components, component certification, among others.

Future works on CBSD would cover open questions for each topic, as follows, according Schneider & Han (Schneider & Han, 2004):

i. **Functionality**: traditionally, an interface specifies the provided services of a component in terms of signatures. An interesting topic to be studied is to discover a way to be able to extract enough information from a component's interface to know how the services of a component must be used and how they can be correctly combined with services of other components;

ii. **Interaction**: an open question to be considered here is about the notion of correctness: when can the composition of components be considered correct? Some possible solutions may be to combine both, a static and dynamic approach, to check the correctness of a composition and take into consideration component environments**;**

iii. **Quality**: the question in this topic is how to verify that the specified quality attributes (non-functional) are met by the given implementation. And also, how the quality attributes of a composition of components are met given the quality attributes of the components involved;

iv. **Management**: in an ideal world, there are components available for any task, so the designed application has only to plug these components together. But how can a suitable component be found given a set of requirements? Nowadays, it is generally not possible to search for components given a description in the problem domain. Many approaches have been thought of to implement component repositories (standardization efforts, suitable ontologies, meta-level descriptions etc.), but there is still a lot of undiscovered land to investigate;

v. **Evolution and Tools**: there are many problems with code maintenance in composed systems (i.e., updating both source code and the corresponding behavioral specification). Hence, here is a definite need for tool support, so that specification and verification can be seamlessly integrated into the development process. As the state-of-the-art is not very advanced, yet, this seems to be a promising area for further exploration; and

vi. **Methodology**: this is probably the area where the least progress was made. There are not any development methodologies specifically tailored towards component-based development yet. Szyperski even claims that "*all facets of software engineering and the entire underlying approach need to be rethought*"

(Szyperski, 2002). Hence, there is still a lot of work to do to come up with a more systematic approach for component-based software engineering.

So, according to Schneider & Han (Schneider & Han, 2004) ideas, it can be concluded that the industry and academy still have not gotten the fundamentals of component-based Software Engineering right, as long as a well-defined and generally-accepted answer to the following questions cannot be given: "*what is a software component*?" and "*how do I correctly compose software components?*". Then, there is little hope for them to reach a level of maturity of the discipline that is acceptable for all stakeholders.

3.8 Chapter Summary

In this chapter, we presented the main topics about software reuse and CBSD according to different authors. The Sections 3.1.1 and 3.1.2 showed, respectively, the context and motivation about using CBSD in the development of software systems. In the Section 3.1.3, components definitions were shown according to Sametinger (Sametinger, 1997), Heineman & Councill (Heineman & Councill, 2001) and Szyperski (Szyperski, 2002). The last two definitions have been treated with details in the Sections 3.1.3.1 and 3.1.3.2, respectively. In the subsequent sections, six component attributes were shown according to Sametinger as follows: functionality, interactivity, concurrency, distribution, forms of adaptation and quality control. After that, Section 3.1.5 showed some forms of components composition followed by Section 3.1.6 which showed two different taxonomies for organizing components.

This chapter also considered topics about components and CBSD such as: component models to provide standards for component implementation interoperability with the most commonly know COM+, CORBA, Java Beans and Enterprise Java Beans (Section 3.2); the involved risks and challenges when components are being used (Section 3.3); how could component markets be organized to have success (Section 3.4) and the existent inhibitors for creating such markets (Section 3.5); CBD methods, especially *Catalysis* and *UML Components* (Section 3.6); and the current state of CBSD including the research directions for the future in this area (Section 3.7).

The next Chapter presents an important study on the software reuse processes area discussing the main processes available and their strong and weak points.

References

(Almeida, 2003) Almeida, E. S. **An Approach for Distributed Component-Based Software Development** *(in portuguese)*. MSc. Dissertation. Federal University of São Carlos, Brazil, 2003.

(Alvaro et al., 2005a) Alvaro, A.; Almeida, E. S.; Meira, S. R. L. **A Component Quality Model for Component Quality Assurance**. In: *Fifth Workshop on Component-Based Development, Short-Paper*. Juiz de Fora, Brazil, 2005.

(Alvaro et al., 2005b) Alvaro, A.; Almeida, E. S.; Meira, S. R. L. **Software Component Certification: A Survey**. In: *31st IEEE EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA), Component-Based Software Engineering Track*. Portugal, IEEE Press, 2005.

(Alvaro et al., 2005c) Alvaro, A.; Almeida, E. S.; Meira, S. R. L. **Towards a Software Component Quality Model**. In: *The 31st IEEE EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*. Work in Progress Session, Porto, Portugal. 2005.

(Bachmann et al., 2000) Bachmann, F.; Bass, L.; Buhman, C.; Comella-Dorda, S.; Long, F.; Robert, J.; Seacord, R.; Wallnau, K. **Technical Concepts of Component-Based Software Engineering 2nd Edition**. Carnegie Mellon Software Engineering Institute, May, 2000.

(Cheeseman & Daniels, 2001) Cheeseman, J.; Daniels, J. **UML Components: A Simple Process for Specifying Component-Based Software**. Addison-Wesley, 2001.

(Codase, 2005) **Codase - Source Code Search Engine**. Available on http://www.codase.com/, Consulted in October, 2005.

(D'Souza & Wills, 2001) D'Souza, D.; Wills, A. C. **Objects, Components, and Frameworks with UML - The Catalysis Approach**. Addison-Wesley, 2001.

(Fractal, 2005) **The Fractal Project**. Available on http://fractal.objectweb.org/, Consulted in October, 2005.

(Gamma et al., 1995) Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. **Design Patterns: Elements of Reusable Object-Oriented Software Development**. Addison-Wesley Professional, 1995.

(Gold & Mohan, 2003) Gold, N.; Mohan, A. **A Framework for Understanding Conceptual Changes in Evolving Source Code**. In: *19th IEEE International Conference on Software Maintenance (ICSM'03)*. IEEE Press, pp. 431, September, 2003.

(Google Code, 2005) **Google Code**. Available on http://code.google.com/, Consulted in October, 2005.

(Harry, 1997) Harry, A. **Formal Methods Fact File: VDM and Z**. Wiley, 1997.

(Heineman & Councill, 2001) Heineman, G. T.; Councill, W. T. **Component-based Software Engineering: Putting the Pieces Together**. Addison-Wesley, 2001.

(Kruchten, 1999) Kruchten, P. **Rational Unified Process: An Introduction**. Addison-Wesley, 1999.

(McIlroy, 1968) McIlroy, M. D. **Mass Produced Software Components**, In: NATO Software Engineering Conference Report, Garmisch, Germany, October, 1968, pp. 79-85.

(Microsoft, 1995) Microsoft Corporation. **The Component Object Model Specification**. 1995.

(Monson-Haefel, 1999) Monson-Haefel, R. **Enterprise JavaBeans**. O'Reilly & Associates, Inc., First edition, June, 1999.

(Nascimento, 2005) Nascimento, L. M. **Component-Based Development in J2ME: A Framework for Graphical Interface Development in Mobile Devices**, (*in portuguese*), BSc. Dissertation, Federal University of Pernambuco, Recife, Pernambuco, Brazil, August, 2005.

(Oberleitner & Gschwind, 2003) Oberleitner, J.; Gschwind, T. **The Vienna Component Framework - Enabling Composition Across Component Models**. In: *25th International Conference on Software Engineering (ICSE'03)*. IEEE Press, pp. 25, May, 2003.

(OMG, 2001) Object Management Group (OMG). **The Common Object Request Broker: Architecture and Specification**. 2.4 edition, 2001.

(OMG, 2002) Object Management Group (OMG). **CORBA Components**. 3.0 edition, 2002.

(Ommering et al., 2000) Ommering, R. V.; Linden, F. V. D.; Kramer, J.; Magee, J. **The Koala Component Model for Consumer Electronics Software**. In: *Computer*. IEEE Press, Vol. 33, No. 3, March, 2000, pp. 78-85.

(Ovum, 2005) **Ovum Componentware: building it, buying it, selling it**. Available on http://www.ovum.com, Consulted in October 2005.

(PECOS, 2005) **PECOS Project**. Available on http://www.pecos-project.org/, Consulted in June, 2005.

(Prieto-Diaz, 1990) Prieto-Diaz, R. **Domain Analysis: An Introduction**. In: *ACM SIGSOFT Software Engineering Notes,* Vol. 15, No. 02, pp. 47-54, April, 1990.

(Ravichandran & Rothenberger, 2003) Ravichandran, T.; Rothenberger, M. A. **Software Reuse Strategies and Component Markets**. In: *Communications of the ACM*. Vol. 46 No. 8, pp. 109-114, August, 2003.

(Ropponen & Lyytinen, 2000) Ropponen J.; Lyytinen, K. **Components of Software Development Risk: How to Address Them? A Project Manager Survey**. In: *IEEE Transactions on Software Engineering*. Vol. 26 No. 2, pp. 98-112, February, 2000.

(Sametinger, 1997) Sametinger, J. **Software Engineering with Reusable Components**. Springer-Verlag, 1997.

(Schneider & Han, 2004) Schneider, J.; Han, J. **Components – the Past, the Present, and the Future**. Swinburne University of Technology, School of Information of Technology, June, 2004.

(Stojanovic et al., 2001) Stojanovic, Z.; Dahanayake, A.; Sol, H. **A Methodology Framework for Component-Based System Development Support**. In: *The 6th CaiSE/IFIP8.1 International Workshop on Evaluation of Modeling*

Methods in Systems Analysis and Design EMMSAD'01. Interlaken, Switzerland, June, 2001.

(Sun Microsystems, 1997) Hamilton, G. **JavaBeans**. Sun Microsystems, http://java.sun.corn/beans/. July, 1997.

(Sun Microsystems, 2001) DeMichiel, L. G.; Yal L. El. C.; Krishnan, S. **Enterprise JavaBeans Specification, Version 2.0**. Sun Microsystems, Proposed Final Draft 2. April 2001.

(Szyperski, 2002) Szyperski, C. **Component Software – Beyond Object-Oriented Programming**. Addison-Wesley, 2002.

(Tanenbaum, 2002) Tanenbaum, A. S. **Distributed Systems: Principles and Paradigms**. Prentice Hall, 2002.

(Traas & Hillegersberg, 2000) Traas, V.; Hillegersberg, J. V. **The software component market on the internet current status and conditions for growth**. In: *ACM SIGSOFT Software Engineering Notes*, Vol. 25 No. 01, January, 2000.

(Turowski, 2003) Turowski, K. **Drivers and inhibitors to the development of a software component industry**. In: *Proceedings of the 29th EUROMICRO Conference "New Waves in System Architecture"*. IEEE Computer Society Press, 2003.

(Vitharana, 2003) Vitharana, P. **Risks and Challenges of Component-Based Software Development**. In: *Communications of the ACM*, Vol. 46 No. 08, pp. 67-72, August, 2003.

(Weinreich, 1997) Weinreich, R. **A Component Framework for Direct Manipulation Editors**. In: *Proceedings of the 25th International Conference on the Technology of Object-Oriented Languages and Systems (TOOLS)*. IEEE Computer Society Press, Melbourne, Australia, 1997.

Chapter 4

Software Reuse Processes: The State-of-The-Art

A process is a collection of related tasks leading to a product (Ezran et al., 2002). Processes are nested: they may be broken down into sub-processes until reaching atomic tasks (sometimes called activities, performed uninterruptedly by one person in one place).

Processes are important and necessary to define how an organization performs its activities, and how people work and interact in order to achieve their goals. In particular, processes must be defined in order to ensure efficiency, reproducibility and homogeneity.

Software processes refer to all of the tasks necessary to produce and manage software, whereas reuse processes are the subset of tasks necessary to develop and reuse assets.

The adoption of either a new, well-defined, managed software process or a customized one is a possible facilitator for success in reuse programs. However, the choice of a specific software reuse process is not a trivial task, because there are technical and non-technical aspects that must be considered.

This chapter presents eleven software reuse processes representing the state-of-the-art of the area and discusses the requirements and important issues that effective reuse processes must consider.

4.1 Introduction

Over the years, several research works including company reports (Endres, 1993), (Bauer, 1993), (Griss, 1994), (Griss, 1995), (Joos, 1994), informal research (Frakes & Isoda, 1995) and empirical studies (Rine, 1997), (Morisio et al., 2002), (Rothenberger et al., 2003) have shown that an effective way to obtain the benefits of software reuse is to adopt a reuse process. However, the existing reuse processes present crucial problems such as gaps in important activities like development *for* and *with* reuse, and lack of emphasis on some specific activities (analysis, design and implementation). Even today, with the ideas of software product lines, there is still no clear consensus about the activities to be performed (inputs, outputs, artifacts) and the requirements that an effective reuse process must meet.

In this context, we agree with Bayer et al. when they said (Bayer et al., 1999): "*Existing methods have been either not flexible enough to meet the needs of various industrial situations, or they have been too vague, not applicable without strong additional interpretation and support. A flexible method that can be customized to support various enterprise situations with enough guidance and support is needed*" (pp. 122).

Motivated by this, the reminder of this chapter presents a detailed review based on eleven software reuse processes representing the state-of-the-art of the area.

4.2 Software Reuse Processes: A Survey

The argument for defining processes for specific software development tasks should be a familiar one. A well-defined process can be observed and measured, and thus improved. A process can be used to capture the best practices for dealing with a given problem. The adoption of processes also allows a fast dissemination of effective work practices. An emphasis on processes helps software development to become more like engineering, with predictable time and effort constraints, and less like art (Rombach, 2000).

A software reuse process, besides presenting issues related to non-technical aspects, must describe two essential activities: the development *for* reuse and the development *with* reuse. In the literature, several research studies aimed at finding efficient ways to develop reusable software can be found. These works focus on two directions: domain engineering and, currently, product lines, as can be seen in the next sections.

4.2.1 Domain Engineering Processes

Domain engineering is the activity of collecting, organizing, and storing past experience in building systems or parts of systems in a particular domain in the form of reusable assets, as well as providing an adequate means for reusing these assets when building new systems (Czarnecki & Eisenecker, 2000).

Among the works of the early 80's and 90's, such as (Neighbors, 1980), (STARS, 1993), (Simos et al., 1996), (Jacobson et al., 1997), (Griss et al., 1998), (Kang et al., 1998), a special effort is put into the domain engineering processes when developing reusable software.

4.2.1.1 The Draco Approach

An example of this area of research may be seen in (Neighbors, 1980). In this work, Neighbors proposed the *first domain engineering approach*, as well as a prototype - Draco - based on transformation technology. The main ideas introduced by Draco include: *Domain Analysis*, *Domain-Specific Languages*, and *Components* as sets of transformations.

Draco supports the organization of software construction knowledge into a number of related domains. Each Draco domain encapsulates the requirements and different implementations of a collection of similar systems.

Neighbors' work has made an important contribution to the domain engineering field, presenting concepts such as generative programming, transformation systems and components. Nevertheless, his approach is very difficult to apply in the industrial environment due the complexity of performing activities such as writing transformations and using the Draco machine. Even with some advances related to his work (Leite, 1994), many of these problems still remain unsolved.

4.2.1.2 Conceptual Framework for Reuse Processes (CFRP)

In this context, in 1992 (STARS, 1993), the Software Technology for Adaptable, Reliable Systems (STARS) program developed the Conceptual Framework for Reuse Processes (CFRP) as a vehicle for understanding and applying the STARS domain-specific reuse-based software engineering paradigm. The CFRP established a framework for considering reuse-related software engineering processes, how they interrelate, and how they can be integrated with each other and with non-reuse-related processes to form reuse-oriented life-cycle process models that can be tailored to organizational needs.

However, CFRP by itself was a very generic framework and organizations will find it too generic to serve as the sole basis for developing a tailored domain-specific reuse-based software engineering life cycle model. Thus, the ROSE Process Model (ROSE PM) (STARS, 1993) was developed by the Paramax STARS team specifically to bridge the gap between the high-level, generic framework and the detailed, prescriptive methods.

ROSE PM is divided into four iterative sub-models: *Organization Management, Domain Engineering, Asset Management,* and *Application Engineering*. *Organization Management* consists of the highest-level organizational Plan, Enact, and Learn activities. In the ROSE model, the project (Domain Engineering, Asset Management, and Application Engineering) activities are hierarchically placed within the Enact portion of Organization Management. The goal of the ROSE *Domain Engineerin*g project sub-model is to capture, organize and represent knowledge about a domain and produce reusable assets that can be further applied to produce a family of systems encompassing that domain. In the *Asset Management* sub-model the goal is to acquire, evaluate, and organize assets produced by domain engineering, and make those assets available as a managed collection that serves as mediation or brokering mechanisms between asset creators and asset consumers. Finally, in the *Application Engineering* the goal is to develop, reengineer, maintain, or evolve software systems, making use of assets created in the domain engineering sub-model.

The key objective of the ROSE model is to integrate software maintenance and reengineering in the context of domain-specific reuse to produce a general reuse-oriented approach to software evolution. However, the main problem with ROSE is that the process model presents the activities in a generic way without specifying, in details, how they should be performed. For example, in *Domain Engineering*, there is an activity defined as *Develop Software Components*, but the steps to perform it are not defined. The same problem happens with other activities such as *Develop Software Architecture* and *Develop Applications Generators*.

4.2.1.3 Organization Domain Modeling (ODM)

Four years after the beginning of the efforts of the STARS project, Mark Simos (Simos et al., 1996) and his group developed the Organization Domain Modeling (ODM) method. Their motivation was that there was a gap in the domain engineering methods and processes available to reuse practitioners. Simos' thesis was that because work in domain analysis for software reuse emerged out of the software engineering research community, many of the methods were developed by technologists. Moreover, Simos highlighted that people trying to launch small-scale domain engineering projects have to negotiate technical and non-technical issues in planning, managing, and transitioning their work. Yet, they usually do not have the support of an already existing reuse infrastructure within the organization.

In fact, the motivation for organizations to adopt reuse practices on a broader scale might well depend on successful results from initial pilot projects. Thus, the ODM method was an attempt to meet this need and establish a foundation for evolving initial pilot projects into reuse programs.

The ODM process consists of three main phases: *Plan Domain*, *Model Domain*, and *Engineer Asset Base*. The purpose of the *Plan Domain* phase is to set objectives and select and scope a domain for the project in alignment with overall organizational needs. In the *Model Domain* phase, the goal is to produce a domain model for the selected domain, based on the domain definition produced in the previous phase. Finally, in the *Engineer Asset Base* phase, the purpose is to scope, architect, and implement an asset base that supports a subset of the total range of variability encompassed by the domain model. *Plan Domain* and *Model Domain* correspond to domain analysis. *Engineer Asset Base* comprehends both domain design and implementation.

As in ROSE, although ODM encompasses the steps of domain engineering, it does not present specific details on how to perform many of its activities. However, this limitation is clearly described in the ODM guidebook. According to Simos et al. the method provides a "*general, high-level guidance in tailoring the method for application within a particular project or organization*" (pp. 01). In this way, critical activities are not properly guided, which may contribute to possible problems for organizations when using the method.

Until Simos' work, the research involving software reuse processes was too general and did non present concrete techniques to perform tasks such as architecture and component modeling and implementation. Consequently, it was difficult to develop reusable object-oriented software using these processes.

4.2.1.4 Reuse-driven Software Engineering Business (RSEB) and FeatuRSEB

Motivated by this problem, three software development experts - Jacobson, Griss and Jonsson - created the Reuse-driven Software Engineering Business (RSEB) (Jacobson et al., 1997). RSEB is a use-case driven systematic reuse process based on UML notation. The method was designed to facilitate both the development of reusable object-oriented software and software reuse. Similar to the Unified Process (Jacobson et al., 1999), RSEB is also iterative and use-case centric.

Key ideas in RSEB are: the explicit focus on modeling variability and maintaining traceability links connecting representation of variability throughout the models, i.e., variability in use cases can be traced to variability in the analysis, design, and implementation object models.

RSEB has separated processes for *Domain Engineering* and *Application Engineering*. *Domain Engineering* in RSEB consists of two processes: *Application Family Engineering*, concerned with the development and maintenance of the overall layered system architecture and *Component System Engineering*, concerned with the development of components for the different parts of the application system with a focus on building and packaging robust, extendible, and flexible components.

In RSEB, *Application Engineering* is composed of steps to: Capture requirements, Perform robustness analysis, Design, Implement, Test and Package the application system.

Despite the RSEB focus on variability, the process components of Application Family Engineering and Component System Engineering do not include essential domain analysis techniques such as domain scoping and feature modeling. Moreover, the method does not describe a systematic way to perform the asset development as proposed. Another shortcoming of RSEB is the lack of feature models. In RSEB, variability is expressed at the highest level in the form of variation points, which are then implemented in other models using variability mechanisms.

Based on the limitations observed during the RSEB utilization, Griss et al. developed FeatuRSEB (Griss et al., 1998), which is a result of integrating FODAcom, an object-oriented adaptation of FODA (Kang et al., 1990) for the telecom domain, into RSEB, through cooperation between Hewlett-Packard and Intecs Sistemi.

FeatuRSEB extends RSEB in two areas: (i) the activity that corresponds to Domain Analysis was extended with steps to perform domain scoping (briefly) and feature modeling and (ii) feature models are used as the main representation of commonality, variability, and dependencies.

Although FeatuRSEB had presented important considerations related to domain analysis, such as the process of extracting functional features from the domain use case model, for example, the main limitations discussed in FeatuRSEB were not solved.

4.2.1.5 Feature-Oriented Reuse Method (FORM)

Concurrently with Griss et al.'s work, a respected researcher in the domain analysis area, Kyo Kang, in conjunction with his colleagues, presented the thesis that there were many attempts to support software reuse, but most of these efforts had focused on two directions: *exploratory research* to understand issues in domain specific software architectures, component integration and application generation mechanisms; and *theoretical research* on software architecture and architecture specification languages, development of reusable patterns, and design recovery from existing code. Kang et al. considered that there were few efforts to develop systematic methods for discovering commonality and using this information to engineer software for reuse. This was their motivation to develop the Feature-Oriented Reuse Method (FORM) (Kang et al., 1998), an extension of their previous work (Kang et al., 1990).

FORM is a systematic method that focuses on capturing commonalities and differences of applications in a domain in terms of features and using the analysis results to develop domain architectures and components. In FORM, the use of features is motivated by the fact that customers and engineers often speak of product characteristics in terms of features the product has and/or delivers. They communicate requirements or functions in terms of features and, for them, features are distinctively identifiable functional abstractions that must be implemented, tested, delivered, and maintained.

The FORM method consists of two major engineering processes: *domain engineering* and *application engineering*. The *domain engineering* process consists of activities for analyzing systems in a domain and creating reference architectures and reusable components based on the analysis results. The reference architectures and reusable components are expected to accommodate the differences as well as the commonalities of the systems in the domain. The *application engineering* process consists of activities for developing applications using the artifacts created during domain engineering.

There are three phases in the *domain engineering process*: *context analysis, domain modeling,* and *architecture (and component) modeling*. This method does not consider the context analysis phase, however the domain modeling phase is very well explored with regards to features. The core of FORM lies in the analysis of domain features and use of these features to develop reusable domain artifacts. The domain architecture, which is used as a reference model for creating architectures for different systems, is defined in terms of a set of models, each one representing the architecture at a different level of abstraction. Nevertheless, aspects such as specification, design, implementation and packaging of the components are explored a little.

After the domain development, the application engineering process is performed. Once again, the emphasis is in analysis phase with the use of the developed features. However, few directions are defined to select the architectural model and develop the applications using the existing components.

4.2.2 Product Line Processes

Until 1998, the software reuse processes were only related to domain engineering issues. However, in this same period, a new trend started to be explored: the software product line area (Clements & Northrop, 2001). Software product lines began to be seen as one of the most promising advances for efficient software development. However, until the late 90's there were few available guidelines or methodologies to develop and deploy product lines beyond existing domain engineering approaches.

On the other hand, domain engineering approaches have not proved to be as effective as expected. According to Bayer et al. (Bayer et al., 1999), there are basically three reasons for this: *misguided scoping of application area, lack of operational* *guidance, and overstressed focus on organizational issues*. Moreover, domains have proved to be difficult to scope and engineer from an enterprise standpoint because a domain captures many extraneous elements that are of no interest to an enterprise. Hence, the domain view provides little economic basis for scoping decisions. Instead, enterprises focus on particular products (existing, under development, and anticipated). This difference in focus is essential for practically supporting the product-driven needs of enterprises.

4.2.2.1 Product Line Software Engineering (PuLSE)

Based on the mentioned limitations of domain engineering and lack of product-lines approaches, Bayer et al. proposed the Product Line Software Engineering (PuLSE) methodology. The methodology was developed with the purpose of enabling the conception and deployment of software product lines within a large variety of enterprise contexts. One important feature of PuLSE is that it is the result of a bottom-up effort: the methodology captures and leverages the results (the lessons learned) from technology transfer activities with industrial customers.

PuLSE is composed of three main elements: *the deployment phases, the technical components, and the support components*. The *deployment phases* are a set of stages that describe activities for initialization, infrastructure construction, infrastructure usage, and evolution and management of product lines. The *technical components* provide the technical know-how needed to make the product line development operational. At the end, the *support components* are packages of information, or guidelines, which enable a better adaptation, evolution, and deployment of the product line.

The PuLSE methodology presents an initial direction to develop software product lines. However, some points are not well discussed. For example, the component PuLSE-DSSA supports the definition of a domain-specific software architecture, which covers current and future applications of the product line. Nevertheless, aspects such as specification, design, and implementation of the architecture's components are not presented. Bayer et. al. consider it an advantage, because PuLSE-DSSA dos not require a specific design methodology nor a specific Architecture Description Language (ADL). We do not agree with this vision because, in our opinion, lack of details is the biggest problem related to software reuse processes.

The same problem can be seen in the Usage phase, in which product line members are specified, derived and validated without explicit details on how this can be done.

According to Atkinson et al. (Atkinson et al., 2000), PuLSE has been applied successfully in various contexts for different purposes. Among the benefits, it has proved to be helpful for introducing sound documentation and development techniques into existing development practices. However, in circumstances where there were no pre-existing processes or well-defined products, the introduction of PuLSE turned out to be problematic. In such cases, the customization of PuLSE was actually more concerned with the introduction of basic software engineering processes than with the adaptation of the product line ideas to existing processes.

4.2.2.2 KobrA approach

From this perspective, the KobrA approach was proposed (Atkinson et al., 2000). The approach represents a synthesis of several advanced software engineering technologies, including product lines, component-based development, frameworks, and process modeling.

KobrA has two main activities: initially, *framework engineering* creates and maintains a generic framework that embodies all product variants that make up the family, including information about their common and disjoint features. Next, *application engineering* uses the framework built during framework engineering to construct specific applications within the domain covered by the framework. In KobrA, a framework is the static representation of a set of Komponents (KobrA component) organized in the form of a tree. Each Komponent is described at two levels of abstraction: a specification, which defines the Komponent's externally visible properties and behavior, and a realization, which describes how the Komponent fulfils the contracts with other Komponents.

Even supporting the development *for* and *with* reuse, KobrA presents some drawbacks. For example, during the *framework engineering* activity, it does not present guidelines to perform tasks such as domain analysis and domain design. This omission is a little surprising, since previous approaches, such as PuLSE, clearly supported activities for planning and scoping.

4.2.2.3 Component-Oriented Platform Architecting Method (CoPAM)

At the same time, another important and different point of view related to product line processes was proposed in (America et al., 2000). According to America et al., software reuse processes are strongly related to non-technical aspects such as the business and the organizational constraints. America et al. mention the case of the Philips company, where several product families (lines) are developed, ranging from consumer electronics to professional systems. For each of these families, a specifically customized family engineering method is necessary, because the business and organization constraints differ. Based on this motivation, the Component-Oriented Platform Architecting Method (CoPAM) Family for Product Family Engineering was proposed. Thus, for each product family, the CoPAM approach advocates the development of a specific family engineering method from the method family, in a step called method engineering.

According to America et al., the reason that they did not consider an alternative approach that would organize the different product families into one larger family (population) was that a large population only makes sense if the products have enough in common and the organization is able to coordinate the marketing, development, and other aspects of a large population.. Otherwise, it may not be profitable to develop them as a single population.

The CoPAM processes are based on several approaches described in the literature, such as PULSE, RSEB, RUP, and on the authors' experience in the industry. The processes have two main subprocesses: *platform engineering* and *product engineering*. The *platform engineering* process develops a platform, which consists of a number of reusable components. A *product engineering* process develops products using these platform components, adding new components where necessary. Both of these receive guidance from and provide feedback to the family engineering process. Apart from that, platform and product engineering processes can be organized along the same lines as in a regular single product development. The family engineering process is different from the other kinds of subprocesses because it deals with family development.

In CoPAM, there is a single family engineering process per product family, but there can be more than one platform or product engineering subprocesses. The family engineering process has seven steps leading to the detailed component architecture for the family. The steps range from informal domain analysis to the creation of family architecture and support. However, details on how to perform specific tasks, (e.g. domain analysis or family architecture) are not presented.

4.2.2.4 Pervasive Component Systems (PECOS)

Until 2000, the software reuse processes were related to the software engineering domain. However, a collaborative project between industry and academia, starting in October 2000 and finishing in September 2002, applied the reuse concepts within the embedded systems domain (Winter et al., 2002). The Pervasive Component Systems (PECOS) project goal was to enable component-based development for embedded systems. Its motivation was that embedded systems are dificult to maintain, upgrade and customize, and they are almost impossible to port to other platforms. In this context, reuse concepts, mainly component-based development, could present relevant improvements.

The PECOS process focuses on two central issues: the first question is *how to enable the development of families of PECOS devices*. The second issue concerns the *composition of a PECOS field device from an asset, pre-fabricated components, and how these components have to be coupled*, not only on the functional, but also on the non-functional level.

The development of PECOS components is performed during application composition when the need for such components is identified and the components cannot be provided from the component repository. Component development is achived in five tasks: *Requirements elicitation, Interface specification, Implementation, Test & Profiling,* and *Documentation & Release*. An interesting task of this cycle is Profiling, which collects detailed non-functional information about the components. The application development is composed of activities to identify, query, select and compose applications that reuse components.

Even presenting an important contribution to the embedded systems domain, the PECOS process has some gaps and drawbacks. For example, in component development, there is no guidance on how the requirements elicitation is performed, how the contracts amongst the components are specified, and how the components are identified. Moreover, the components are not developed in tasks such as domain analysis. In PECOS, the components are contructed in an isolated way, without this view.

4.2.2.5 Form's extension

The most current process related to product line is seen in (Kang et al., 2002). In this work, Kang et al. present an evolution of their previous work, FORM (Kang et al., 1998). FORM was extended in order to support the development of software product lines.

FORM's product lines consist of two processes: *asset development* and *product development*. Asset development consists of analyzing a product line (such as marketing and product plan development, feature modeling, and requirements analysis) and developing architectures and reusable components based on analysis results.

Product development includes analyzing requirements, selecting features, selecting and adopting architecture, adapting components and, finally, generating code for the product.

The major contribution of this work, when compared to FORM is the concern for business aspects such as the 'marketing and product plan' (MPP). MPP's goal is to identify the information to be gathered during the marketing and business analyses. It includes a market analysis, a marketing strategy, product features, and product feature delivery methods.

On the other hand, the process lacks details on specific activities such as conceptual architecture design and architecture refinement. In these activities, there is no information on how to identify the components and specify dependencies between them. Moreover, the process of product development and its activities are not discussed.

4.3 Towards an Effective Software Reuse Process

Figure 4.1 summarizes the timeline of research on the software reuse processes area, in which we mark (with "X") the milestones of research in the area. Initially, Neighbors (Neighbors, 1980) proposed the first approach for domain engineering, as well as a prototype based on transformation technology. Next, Simos et al. presented ODM (Simos et al., 1996), a systematic approach for Domain Engineering developed in STARS project.

In 1997, Jacobson, Griss and Jonsson created the RSEB, the first use-case driven systematic reuse process based on the UML notation. The method was designed to facilitate both the development of reusable object-oriented software and software reuse using ideas on iterative and use-case centric development. Two years later, the idea of software product lines was first mentioned related to a reuse process with the development of PuLSE (Bayer et al., 1999). Next, based on the PuLSE methodology, Atkinson et al. presented the KobrA approach.

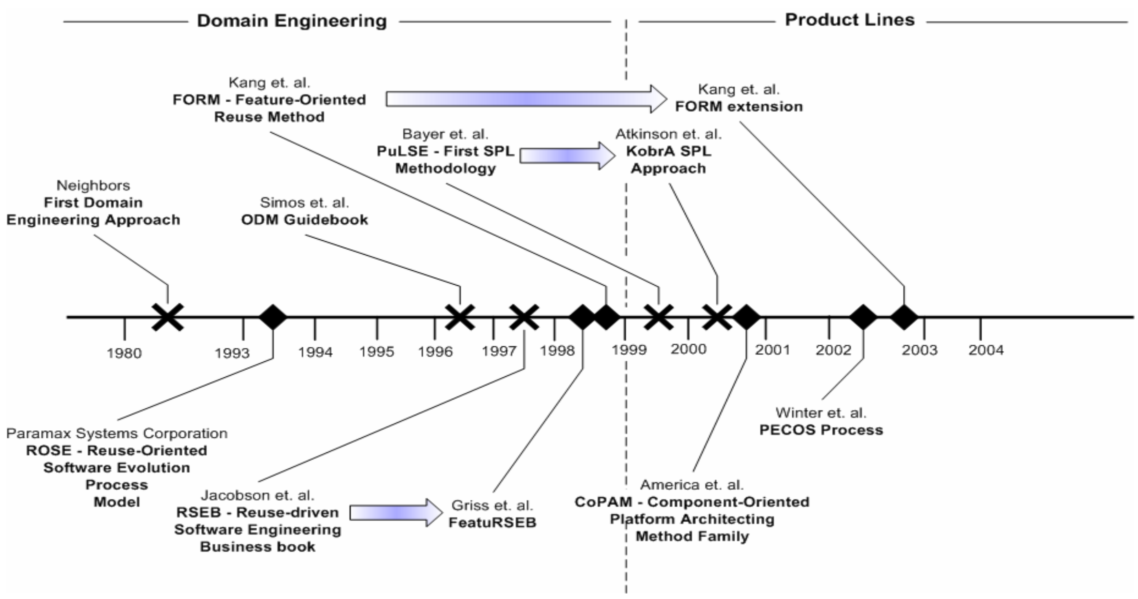

Domain Engineering
Product Lines
Kang et. al.
FORM - Feature-Oriented Reuse Method
Kang et. al.
FORM extension
Bayer et. al.
PuLSE - First SPL Methodology
Atkinson et. al.
KobrA SPL Approach
Neighbors
First Domain Engineering Approach
Simos et. al.
ODM Guidebook
1980
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
Paramax Systems Corporation
ROSE - Reuse-Oriented Software Evolution Process Model
Jacobson et. al.
RSEB - Reuse-driven Software Engineering Business book
Griss et. al.
FeatuRSEB
America et. al.
CoPAM - Component-Oriented Platform Architecting Method Family
Winter et. al.
PECOS Process

Figure 4.1. Research on software reuse processes timeline.

Currently, the software reuse processes are related to software product line issues. However, some important questions, initially discussed in domain engineering processes and nowadays in product line processes, still remain without clear answers (How to perform efficient development *for* reuse? How to solve the gap among analysis, design and implementation in reuse processes? How to achieve development *with* reuse in conjunction with development *for* reuse?).

In the next sections, we try to answer these questions by presenting a set of requirements for an effective software reuse process. We are aware that this is not a definitive set. Nevertheless, we believe that the identified requirements constitute a solid basis for future work.

4.3.1 Development *for* reuse

Design for black-box reuse is primarily a specification issue and is determined by the generality of its implementation. Design for white-box reuse, on the other hand, is primarily determined by design issues, such as modularity, simplicity, and structuredness. Current design lifecycles do not make this distinction. Moreover, the development *for* reuse processes presents gaps among the phases of analysis, design, and implementation. According to Szyperski (Szyperski, 2002), assets such as components, for example, are already a reality at the implementation level, and now this concept must be found at the earlier phases of the development lifecycle. In doing so, reuse principles and concepts should be consistently applied throughout the development process, and consistently followed from one phase to the other.

4.3.2 Development *with* reuse

The development *with* reuse activity is also very important for software reuse. Methods to identify requirements, instantiate architectures for specific products or applications, make adaptations, and integrate them into the new code are needed. Current reuse processes have presented few advances towards this integration, because the major concern is related to the development *for* reuse.

4.3.3 Metrics

As in any engineering activity, measurement is vital for systematic reuse. In general, reuse benefits (improved productivity and quality) are a function of the achieved reuse level. A reuse process must define what, where, and when to measure. However, metrics are often neglected in existing reuse processes.

4.3.4 Costs

Economic considerations are an important aspect of reuse, since the investments in software reuse must be justified by some economic scale. Nevertheless, even with important works in this area, such as (Poulin, 1997), current reuse processes do not consider cost aspects such as, for example, ways to estimate the cost of development for reuse considering domain engineering or product lines.

4.3.5 Reengineering

Software reengineering is being used to recover legacy systems and allow their evolution (Jacbon & Lindstrom, 1991). It is performed mainly to reduce maintenance costs, and improve development speed and systems readability.

However, the importance of reengineering goes beyond such technical aspects. Legacy systems represent much of the knowledge produced and maintained by an organization, which cannot afford to lose it. Thus, reengineering allows this knowledge to be reused, in the form of reconstructed code and documentation. Chapter 5 presents a detailed discussion on reengineering and its relation to software reuse.

4.3.6 Adaptation

Many organizations are actively pursuing software process maturity in order to increase their competitiveness. Usually, they use measures such as the Capability Maturity Model (CMM) to structure their process improvement initiatives. One method often cited for accelerating process improvement within an organization is to replicate a standard, reusable organizational process within other projects. Hollenbach defined this approach as a reusable process (Hollenbach & Frakes, 1996). According to Hollenbach, process reuse is the use of one process description in the creation of another process description. This should not be confused with multiple executions of the same process on a given project.

However, this method presents some problems when the boundary to reuse the process is not the organization, but other software factories. Moreover, the problem is still bigger when the process in question is a software reuse process. Thus, what is needed is an effective process to capture guidelines that lead to the adaptation of an organization's process to become reuse-centered.

We believe that Hollenbach's initial work can offer some directions to define the aspects of the adaptation process. However, we deem that the formalization of a Reuse Maturity Model (RMM) with levels of reuse and key practices can offer important insights, since organizations can be classified according to RMM levels and the process can be adapted more accurately.

4.3.7 Software Reuse Environment

CASE tools have always helped in software development, providing improved productivity and quality. However, the benefit of CASE is even bigger when in the form of integrated environments, with several tools that add their contributions to the process, guiding the software engineer through the entire development lifecycle.

With software reuse, this could not be different. Inside a Software Reuse Environment, tools are used in the development and reuse of assets. There are some issues that must be considered inside a Software Reuse Environment, such as:

i. Distinction between development *for* reuse and development *with* reuse. These are different activities, which must have distinct tool support;

ii. Tool integration. The tools of the environment must have different levels of integration (Ambriola et al., 1997), including data and presentation integration, but mainly process integration, i.e., they must be focused on a single process, in this case, a reuse process;

iii. Reusability. Not just code, but every kind of asset must be reused, including models, test cases, requirements, among others;

iv. Referential Integrity. Reuse often causes assets to reference each other. In order to avoid confusion, these references must be correctly managed;

v. Software Configuration Management. As in any development environment, the control of the software evolution reduces the risks during development and maintenance, and so, this is a major requirement in a reuse environment; and

vi. Technology and language independence. The environment must not be fixed in a single technology or language. Otherwise, it would be of little use to a software factory, which must be able to use any technology and/or language in its projects.

4.4 Summary of the Study

Table 4.1 shows the relation between the works described in section 2 and the requirements from section 3. Each number corresponds to a work: 1. Neighbors (Draco), 2. ROSE PM, 3. ODM, 4. RSEB, 5. FeatuRSEB, 6. FORM, 7. PuLSE, 8. KobrA, 9. CoPAM, 10. PECOS, 11. FORM's extension. In the table, a white smile indicates that the requirement is fully satisfied by the work. On the other hand, a black smile indicates that the requirement is partially satisfied. Gaps show that the requirement is not even addressed by the work.

Table 4.1. Relation between the works on software reuse processes and the requirements.

Requirements	Works										
	1	2	3	4	5	6	7	8	9	10	11
Development *for* reuse **- Domain Analysis**											
Planning		☻	☻				☻	☻			☻
Scoping		☻	☻				☻	☻			
Modeling	☺	☻	☻		☺	☺	☺	☺	☻		☺
- Domain Design											
Identify components	☺	☻	☻	☻	☻	☺	☻	☺	☻	☻	☺
Specification, Design components	☺	☻	☻	☻	☻	☻	☻	☺	☻	☻	☻
Quality attributes for architecture											
Architecture documentation											
Architecture evaluation		☻	☻								
- Domain Implementation	☺	☻	☻	☻	☻	☻	☻	☺	☻	☻	☻
Development *with* reuse	☺	☻		☻	☻	☻	☻	☻	☻	☻	☻
Metrics											
Costs											
Reengineering											
Adaptation		☻	☻	☻			☻	☻			
Software Reuse Environment											

4.5 Chapter Summary

Adopting a software reuse process is an effective way for organizations to obtain the benefits related to software reuse, such as quality improvements and time-to-market reduction. However, choosing a reuse process is not a trivial task, even considering the division between domain engineering and product lines.

In this chapter, we discussed eleven software reuse processes corresponding to the state-of-the-art of the area. This survey contribution is twofold: it can be seen as a guide to aid organizations in the adoption of a reuse process, and it offers the basis for a formation of a reuse process. Thus, based on the weaknesses and strengths of existing processes, we presented a set of requirements for an effective software reuse process.

So far, we just discussed techniques (Component-Based Development, Domain Engineering, Software Product Lines) that are associated with asset creation without considering the organizational legacy. The next chapter presents an important aspect related to this issue; the reengineering area, which aims, *among others things*, to identify, recover and package assets in a reusable way.

4.6 References

(Ambriola et al., 1997) Ambriola, V.; Conrad, R.; Fuggetta, A. **Assessing Process-Centered Software Engineering Environments**, In: *ACM Transactions on Software Engineering and Methodology (TOSEM)*, Vol. 06, No. 03, July, 1997, pp. 283–328.

(America et al., 2000) America, P.; Obbink, H.; Ommering, R.V.; Linden, F.V.D. **CoPAM: A Component-Oriented Platform Architecting Method Family for Product Family Engineering**, In: *The First Software Product Line Conference (SPLC)*, Kluwer International Series in Software Engineering and Computer Science, Denver, Colorado, USA, 2000, p.15.

(Atkinson et al., 2000) Atkinson, C.; Bayer, J.; Muthig, D. **Component-Based Product Line Development: The KobrA Approach**, In: *The First Software Product Line Conference (SPLC)*, Kluwer International Series in Software Engineering and Computer Science, Denver, Colorado, USA, 2000, p.19.

(Bauer, 1993) Bauer, D. **A Reusable Parts Center**, In: *IBM Systems Journal*, Vol. 32, No. 04, 1993, pp. 620-624.

(Bayer et al., 1999) Bayer, J.; Flege, O.; Knauber, P.; Laqua, R.; Muthig, D.; Schmid, K.; Widen, T.; DeBaud, J. **PuLSE: A Methodology to Develop Software Product Lines**, In: *Symposium on Software Reusability (SSR)*, ACM Press, Los Angeles, USA, May, 1999, pp. 122-131.

(Bergey et al., 1999) Bergey, J.; Smith, D.; Tilley, S.; Weiderman, N.; Woods, S. **Why Reengineering Projects Fail**, In: *Technical Report CMU/SEI-99-TR-010*, April, 1999, p. 37.

(Clements & Northrop, 2001) Clements, P. Northrop, L.M. **Software Product Lines: Practices and Patterns**, *Addison Wesley*, August, 2001, p. 608.

(Czarnecki & Eisenecker, 2000) Czarnecki, K.; Eisenecker, U.W. **Generative Programming: Methods, Tools, and Applications**, *Addison-Wesley*, 2000, p. 832.

(Endres, 1993) Endres, A. **Lessons Learned in an Industrial Software Lab**, In: *IEEE Software*, Vol. 10, No. 05, September, 1993, pp. 58-61.

(Ezran el al., 2002) Ezran M.; Morisio, M.; Tully, C. **Practical Software Reuse**, *Springer*, 2002.

(Frakes & Isoda, 1995) Frakes, W.B.; Isoda, S. **Success Factors of Systematic Software Reuse**, In: *IEEE Software*, Vol. 12, No. 01, January, 1995, pp. 14-19.

(Griss, 1994) Griss, M.L. **Software Reuse Experience at Hewlett-Packard**, In: *Proceedings of the 16th IEEE International Conference on Software Engineering (ICSE)*, Sorrento, Italy, ACM Press, May, 1994, pp. 270.

(Griss, 1995) Griss, M.L. **Making Software Reuse Work at Hewlett-Packard**, In: *IEEE Software*, Vol. 12, No. 01, January, 1995, pp. 105-107.

(Griss et al., 1998) Griss, M.L.; Favaro, J.; d' Alessandro, M. **Integrating Feature Modeling with the RSEB**, In: *The 5th IEEE International Conference on Software Reuse (ICSR)*, Victoria, Canada, June, 1998, pp. 76-85.

(Jacbon & Lindstrom, 1991) Jacobson, I.; Lindstrom, F. **Reengineering of old systems to an object-oriented architecture**, In: *Proceedings of the OOPSLA*, Phoenix, Arizona, USA, 1991. pp. 340–350.

(Jacobson et al., 1997) Jacobson, I.; Griss, M.L.; Jonsson, P. **Reuse-driven Software Engineering Business (RSEB)**, *Addison-Wesley*, 1997, p. 497.

(Jacobson et al., 1999) Jacobson, I.; Booch, G.; Rumbaugh, J. **The Unified Software Development Process**, *Addison-Wesley*, February, 1999, p. 463.

(Joos, 1994) Joos, R. **Software Reuse at Motorola**, In: *IEEE Software*, Vol. 11, No. 05, September, 1994, pp. 42-47.

(Hollenbach & Frakes, 1996) Hollenbach, C.; Frakes, W.B. **Software Process Reuse in an Industrial Setting**, In: *Proceedings of the 4th International Conference on Software Reuse (ICSR)*, Orlando, Florida, USA, 1996, pp. 22-30.

(Kang et al., 1990) Kang, K.C.; Cohen, S.G.; Hess, J.A.; Novak, W.E.; Peterson, A.S. **Feature-Oriented Domain Analysis (FODA) Feasibility Study**, In: *Technical Report CMU/SEI-90-TR-21*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, 1990.

(Kang et al., 1998) Kang, K.C.; Kim, S.; Lee, J.; Kim, K.; Shin, E.; Huh, M. **FORM: A Feature-Oriented Reuse Method with domain-specific reference architectures**, In: *Annals of Software Engineering Notes*, Vol. 05, 1998, pp. 143-168.

(Kang et al., 2002) Kang, K.C.; Lee, J.; Donohoe, P. **Feature-Oriented Product Line Engineering**, In: *IEEE Software*, Vol. 19, No. 04, July/August, 2002, pp.58-65.

(Leite, 1994) Leite, J.C.S.P.; **Draco-Puc: A Technology Assembly for Domain Oriented Software Development**, In: *The Proceedings of the 3th IEEE International Conference on Software Reuse (ICSR)*, Rio de Janeiro, Brazil, November, 1994, pp. 94-100.

(Morisio et al., 2002) Morisio, M.; Ezran, M.; Tully, C. **Success and Failure Factors in Software Reuse**, In: *IEEE Transactions on Software Engineering*, Vol. 28, No. 04, April, 2002, pp. 340-357.

(Neighbors, 1980) Neighbors, J.M. **Software Construction Using Components**, *PhD Thesis*, University of California, Irvine, Department of Information and Computer Science, USA, 1980, p. 75.

(Poulin, 1997) Poulin, J.S. **Measuring Software Reuse: Principles, Practices, and Economic Models**, *Addison-Wesley*, 1997, p. 195.

(Rine, 1997) Rine, D.C. **Success Factors for software reuse that are applicable across Domains and businesses**, In: *ACM Symposium on Applied Computing*, San Jose, California, USA, ACM Press, March, 1997, pp. 182-186.

(Rothenberger et al., 2003) Rothenberger, M.A.; Dooley, K.J.; Kulkarni, U.R.; Nada, N. **Strategies for Software Reuse: A Principal Component Analysis of Reuse Practices**, In: *IEEE Transactions on Software Engineering*, Vol. 29, No. 09, September, 2003, pp. 825-837.

(Rombach, 2000) Rombach, D. **Fraunhofer: The German Model for Applied Research and Technology Transfer**, In: *The Proceedings of the 22nd ACM International Conference on Software Engineering (ICSE)*, Limerick, Ireland, May, 2000, pp. 25-34.

(Simos et al., 1996) Simos, M.; Creps, D.; Klingler, C.; Levine, L.; Allemang, D. **Organization Domain Modeling (ODM) Guidebook**, Version 2.0, *Technical Report*, June, 1996, p. 509.

(STARS, 1993) **Software Technology for Adaptable, Reliable Systems (STARS), The Reuse-Oriented Software Evolution (ROSE) Process Model**, *Technical Report*, July, 1993, p. 143.

(Szyperski, 2002) Szyperski, C. **Component Software: Beyond Object-Oriented Programming**, 2nd Edition, *ACM Press*, 2002, p. 588.

(Winter et al., 2002) Winter, M.; Zeidler, C.; Stich, C. **The PECOS Software Process**, In: *Workshop on Component-based Software Development, in the 7th International Conference on Software Reuse (ICSR),* Austin, Texas, USA, 2002, p.07.

Chapter 5

Software Reengineering and Reverse Engineering: A Survey

The demand for reverse engineering usage has been growing significantly over the last years. The need for different business sectors to adapt their systems for the Web or to use other technologies is stimulating research into methods, tools and infrastructures that support the evolution of existing applications. This chapter analyses the main research on reverse engineering and software reengineering areas, and discusses the requirements for efficient software reengineering aimed at obtaining greater reuse levels, in order to support further software evolutions.

5.1 Introduction

Most organizations stand at a crossroads of competitive survival. A crossroads created by the information revolution that is now shaping/shaking the world. Information Technology has gained so much importance that companies and governments must either step into, taking advantage of modern information processing technology, or lag behind and be bypassed. The point is that most of the critical software systems that companies and government agencies depend upon were developed many years ago, and maintenance is not enough to keep these systems updated, according to technological changes.

Even worse than deprecation, Lehman and Belady empirically prove in (Lehman & Belady, 1985) that, if no improvement is made, maintenance degrades the software quality and, therefore, its maintainability. As the amount of maintenance activity increases, the quality deteriorations, called aging symptoms (Visaggio, 2001), become heavier to manage, turning the legacy system into a burden that cannot be thrown away, and consuming vital resources from its owner.

To solve this problem, Visaggio (Visaggio, 2001) presents some experimental evidence showing that the reengineering process can decrease some aging symptoms. However, even reconstructing the legacy system, there is no guarantee that the new system will be more maintainable than before. For instance, the object-orientation, often considered the "*solution for the software maintenance*" (Bennett & Rajlich, 2000), created new problems for the maintenance (Ossher & Tarr, 1999) and should be used with caution to assure that the maintenance will not be more problematic than in traditional legacy systems.

Current reengineering methods and approaches do not address these problems, leaving the decisions up to the software engineers. This is especially true in the Reverse Engineering phase. This chapter presents a survey of the reengineering and reverse engineering areas, trying to establish a relationship between them, and present a set of requirements that should be dealt with in an effective manner, where reuse and maintainability issues are equally balanced in the reengineered system.

5.2 Reengineering and Reverse Engineering

In general, reengineering is a way to achieve software reuse and to understand the concepts underlying the application domain. Its usage makes it easier to reason about reuse information in analysis and design documents which are not always available in legacy systems.

According to the literature (Sneed, 1995), (Olsem, 1998), (Bennett & Rajlich, 2000) and (Bianchi et al., 2003), reengineering is usually performed with one of the following objectives:

- **Improve maintainability**: maintenance efforts can be reduced through reengineering by producing smaller modules that are easier to maintain. The problem is that it is not simple to measure the impacts of this activity. In fact, it may take years before the maintenance effort reductions can be observed, making it difficult to determine if these benefits were achieved through reengineering or due to other reasons;
- **Migration**: reengineering can be used to move the software to a better or less expensive operational environment. It also may convert old

programming languages into new programming languages, with more resources or higher flexibility;

- **Achieve greater reliability**: the reengineering process includes activities that reveal potential defects, such as redocumentation and testing, making the system more stable and trustable; and
- **Preparation for functional enhancement**: decomposing programs into smaller modules improves their structure and isolates them from each other, making it easier to change or add new functions without affecting other modules.

5.2.1 The terminology

The terminology used in the analysis technologies and legacy systems understanding is presented by Chikofsky and Cross (Chikofsky & Cross, 1990) with the objective of rationalizing terms that are already in use. The main terms are:

- **Reverse Engineering** is the process of analyzing a subject system to identify its components and interrelationships in order to create representations in another form or at a higher abstraction level;
- **Design Recovery** is a subset of reverse engineering, where domain knowledge, external information and deductions are added to the observations of the subject system to identify the high-level abstractions that are beyond the information obtained directly by examining just the system itself;
- **Redocumentation** is the creation or revision of a semantically equivalent representation with the same abstraction level. The products of this process are usually considered alternative views (e.g., dataflow, data structure and control flow) directed to human audience;
- **Restructuring** is the transformation from one representation form to another at the same abstraction level, preserving the subject system's external behavior (functionality and semantics);

- **Forward Engineering** is the traditional software development process, which begins with high-level abstractions and logical, implementation-independent designs, and ends with physical implementations; and
- **Reengineering** is the examination and modification of a subject system to reconstitute it in a new form and its subsequent implementation.

The relationships between these terms are shown in Figure 5.1 (Chikofsky & Cross, 1990). The software life-cycle contains three stages: **(i)** Requirements, **(ii)** Design and **(iii)** Implementation, with different abstraction levels. The **Requirements** stage deals with the problem specification, including objectives, constraints and business rules. The **Design** stage deals with the solution specification. The **Implementation** stage deals with the code, test and delivery of the system in operation.

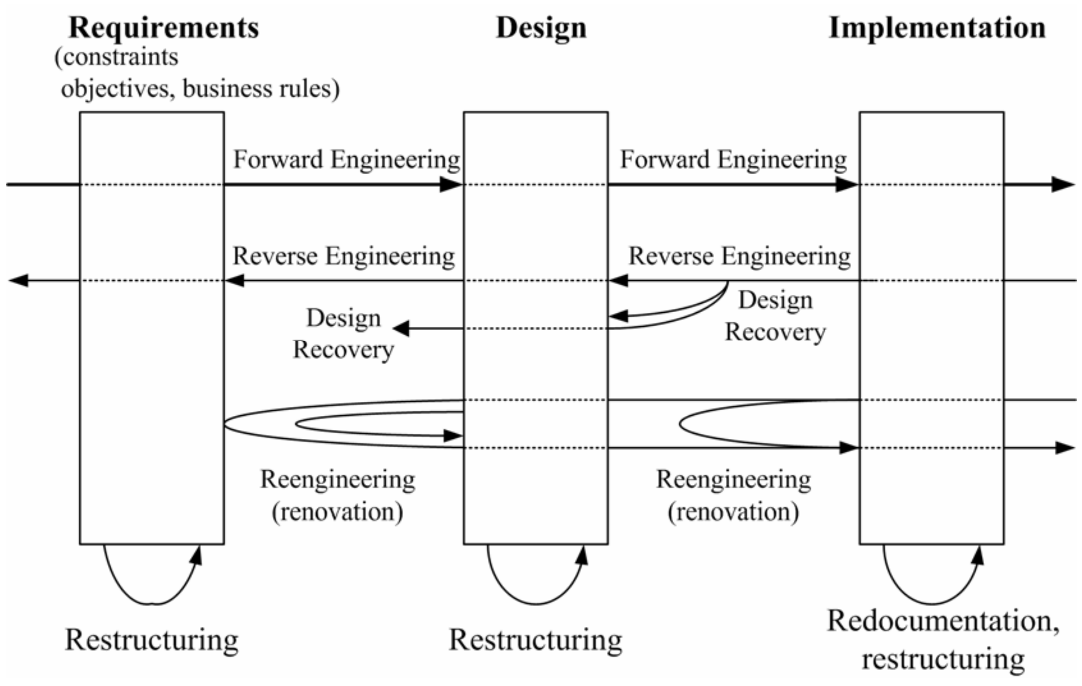


Figure 5.1. Software life cycle (Chikofsky & Cross, 1990)

Figure 5.1 shows the direction followed in the Forward Engineering stage, going to lower abstraction levels. It also shows that Reverse Engineering follows the inverse direction and that Redocumentation can be used to increase the abstraction level. The Reengineering usually includes a Reverse Engineering stage, followed by some form of Forward Engineering or Restructuring.

The most important activity of Reengineering, however, is reverse engineering, since it is responsible for gathering all the information that will be used in the process. It starts from the source code and generates the subject system's design or specification.

5.2.2 Reverse Engineering

Reverse engineering should produce, preferably in an automatic way, documents that help software engineers in understanding the system. Over the last ten years, reverse engineering research has produced a number of capabilities for analyzing code, including subsystem decomposition (Umar, 1997), concept synthesis (Biggerstaff, et al., 1994), design, program and change pattern matching (Gamma et al., 1995), (Stevens & Poley, 1998), analysis of static and dynamic dependencies (Systa, 1999), object-oriented metrics (Chidamber & Kemerer, 1994), and others. In general, these approaches have been successful in treating the software at the syntactic level to address specific information needs and to span relatively narrow information gaps.

The reverse engineering process is illustrated in Figure 5.2 (Sommerville, 1996).

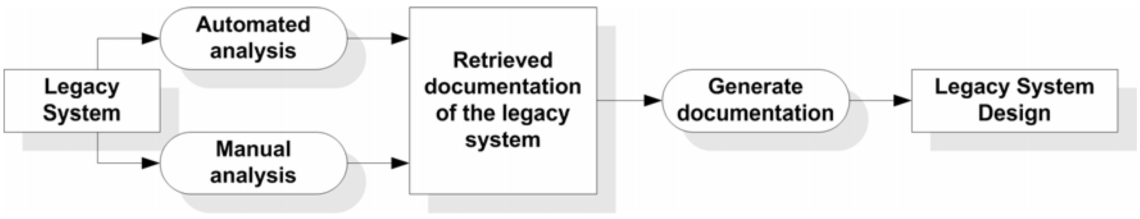


Figure 5.2. Reverse Engineering Process (Sommerville, 1996)

The process starts with the analysis phase. During this phase, the legacy system may be analyzed by tools to obtain information regarding the system (design, specifications or architectural diagrams). The software engineers work in the code level to retrieve this information, which should be then stored, preferably as high abstraction level diagrams.

In industrial-scale systems, reverse engineering is usually applied through semi-automated approaches using CASE (Computer Aided Software Engineer) tools which are used to analyze the semantics of the legacy code. The result of this process is sent to restructuring and rebuilding tools and to the forward engineering activities to complete the reengineering process.

5.3 Software Reengineering and Reverse Engineering Approaches

A great number of techniques and methods have been proposed to face the software reconstruction problem. This section presents a survey on these approaches, with the objective of defining a common set of requirements that should be addressed in an effective approach. There are several research trends on reverse engineering, but five stand out: **(i)** Source-to-Source Translation, **(ii)** Object Recovery and **(iii)** Specification, **(iv)** Incremental Approaches and **(v)** Component-Based Approaches. These trendes are described in the following sections.

5.3.1 Source-to-Source Translation approaches

Essentially, all program translators (both source-to-source translators and compilers) operate via transliteration and refinement. The source program is first transliterated from the source language into the target language on a statement-by-statement basis. Various refinements are then applied in order to improve the quality of the output. Although acceptable in many situations, this approach is fundamentally limited to the reengineering context due to the low quality of the produced output: it tends to be insufficiently sensitive to global features of the source program and too sensitive to irrelevant local details.

The transformation process is controlled by dividing it into a number of phases, which each phase applies transformation selected from a small set. The transformations within each set are chosen in a way that conflicts among transformations will not arise. The Lisp-to-Fortran translator proposed by Boyle (Boyle & Muralidharan, 1984) is based on the transformational approach. The translator handles a subset of Lisp which does not include hard-to-translate features, such as the ability to create and execute new Lisp code. Since readability is not a goal of this translator, the readability of the output is abandoned in favor of producing reasonably efficient Fortran code. As discussed in (Boyle & Muralidharan, 1984), this translator is perhaps better through of as a Lisp to Fortran compiler rather than source-to-source translator.

The main advantage of source-to-source translation is that it is faster than traditional approaches, and it requires less expensive manual effort. However, it is clear that there are still some significant limitations in the quality of the output that is produced by typical translators. This can be clearly seen in source-to-source translation, where human intervention is typically required in order to produce acceptable output.

Waters (Waters, 1988) presents an alternative translation paradigm – abstraction and reimplementation. In this paradigm, the source program is first analyzed in order to obtain a programming-language independent understanding of the computation performed by the program as a whole. Based on this understanding, the program is reimplemented in the target language. In contrast to the standard translation approach of transliteration and refinement, translation via analysis and reimplementation utilizes an in-depth understanding of the source program, which makes it possible for the translator to generate the target code without being constrained by irrelevant details.

5.3.2 Object Recovery and Specification approaches

The buzzword of the late 1980's and early 1990's was the Object-Oriented (OO) paradigm. The OO paradigm offers some desirable characteristics, which significantly help improving software reuse. This is the predominant software trend of the 1990's and it should enhance maintainability, reduce the error rate, increase productivity and make the data processing world a better place to live (Meyer, 1997).

The idea of applying OO reverse engineering is that in a simple way and with limited efforts the software engineer can derive a model from an existing system. With a model, one can better reason about a change to be performed, its extent, and how it shall be mapped onto the existing system. Moreover, the derived model can serve as a basis for a future development plan.

The first relevant work involving the OO technology was presented by Jacobson and Lindstron (Jacobson & Lindstron, 1991), who applied reengineering in legacy systems that were implemented in procedural languages, such as C or COBOL, obtaining OO systems. Jacobson and Lindstron state that reengineering should be accomplished in a gradual way, because it would be impractical to substitute a whole working system for a completely new one (which would demand many resources). They consider three different scenarios: **(i)** changing the implementation without changing functionality; **(ii)** partial changes in the implementation without changing functionality; and **(iii)** changes in the functionality. Object-orientation was used to accomplish this modularization. Jacobson and Lindstron used a specific CASE tool and defend the idea that reengineering processes should focus on tools to aid the software engineer.

Jacobson and Lindstron also state that reengineering should be incorporated as a part of the development process, instead of being a substitute for it.

Gall and Klösh (Gall & Klösh, 1994) propose heuristics to find objects based on Data Store Entities (DSEs) and Non-Data Store Entities (NDSEs) which act primarily over tables representing basic data dependency information. The tables are called (m, u)- tables, since they store information on the manipulation (m) and use (u) of variables. With the help of an expert, a basic assistant model of the OO application architecture is devised for the production of the final generated Reverse Object-Oriented Application Model (ROOAM).

Yeh et al. (Yeh et al., 1995) propose a more conservative approach based not just on the search of objects, but directed towards the finding of Abstract Data Types (ADTs). Their approach, called OBAD, is encapsulated by a tool, which uses a data dependency graph between procedure and structure types as a starting point for the selection of ADT candidates. The procedures and structure types are the graph nodes, and the references between the procedures and the internal fields are the edges. The set of connected components in this graph forms the set of candidate ADTs.

In 1995, Wilkening et al. (Wilkening et al., 1995) present a process for legacy systems reengineering using parts of their implementation and design. The process begins with the preliminary source code restructuring, to introduce some improvements, such as removal of non-structured constructions, "*dead*" code (e.g: code that is not accessible from any of the system's starting points) and implicit types. The purpose of this preliminary restructuring is to produce a program that is easier to analyze, understand and restructure. Then, the produced source code is analyzed and its representations are built in a high abstraction level. Those representations are used in the subsequent restructuring steps, redesign and redocumentation, which are repeated as many times as necessary in order to obtain a fully restructured system. Then, reimplementation and tests are performed, finalizing the reengineering process.

According to Wilkening, the transformation of procedural programs to an OO structure presupposes that the programs are structured and modular, otherwise they cannot be transformed. A program is said to be structured if it has no GOTO-like branches from one code segment to another, for instance, and is said to be modular if it is divided in a hierarchy of code segments, each one with a single entry and a single exit. The segments, or procedures, should correspond to the elementary operations of the program and each operation should be reachable by invoking it from a higher level routine. The other prerequisite for a modular program is that there must be a procedural calling tree for the subroutine hierarchy so that all subroutines are included as part of the procedure which calls or performs them.

Another fundamental work in OO reengineering is presented in (Sneed, 1996), where Sneed describes a tool aided reengineering process to extract objects from existent COBOL programs. Sneed emphasizes the predominance of the object technology, mainly in distributed applications with graphic interfaces, questioning the need to migrate legacy systems to that technology. Sneed identifies obstacles to OO reengineering, such as the object identification, the procedural nature of most of the legacy systems, the code redundancy and the arbitrary use of names.

Similarly to Wilkening et al., Sneed assumes as prerequisites for OO reengineering the structuring and modularization of programs. Sneed also advocates the need for the existence of a system call tree, to identify procedure calls within the system. Sneed's OO reengineering process is composed of five steps: **(i)** object selection, **(ii)** operation extraction, **(iii)** feature inheritance, **(iv)** redundancy elimination and **(v)** syntax conversion. **Step (i)** must be performed by the user, optionally supported by a tool. **Step (ii)** extracts operations performed on the selected objects, replacing the removed code segments with calls to their respective operations. **Step (iii)** creates attributes in the objects, to represent the data accessed by the operations removed in step (ii). **Step (iv)** merges similar classes and removes redundant classes. Finally, **step (v)** converts the remaining classes into Object-COBOL, in a straightforward conversion process.

The transformation of procedural programs into OO programs is not trivial. It is a complex, multi-step, *m:n* transformation process which requires human intervention in defining what objects should be chosen. Although there are already some commercially available tools to help the task of reverse engineering, a still bigger effort is needed to face the complexity of the existing systems, not only because of their size, but also due to their intrinsic complexity.

5.3.3 Incremental approaches

The approaches presented in sections 5.3.1 and 5.3.2 involve the entire system reconstruction as an atomic operation. For this reason, the software must be frozen until the proccess execution has been completed; in other words, no changes are possible during this period. In fact, if a change or enhancement is introduced, the legacy system and its replacing candidate would be incompatible, and software engineers would have to restart the whole process. This situation causes an undesirable loop between the maintenance and the reengineering process.

To overcome this problem, several researchers have suggested wrapping the legacy system, and considering it as a black-box component to be reengineered. Due to the iterative nature of this reengineering process, during its execution the system will include both reengineered and legacy components, coexisting and cooperating in order to ensure the continuity of the system. Finally, any maintenance activities, if required, have to be carried out on both the reengineered and the legacy components, depending on the procedures impacted by the maintenance.

The first important iterative process was proposed in (Olsem, 1998). According to Olsem, legacy systems are formed by four classes of components (Software/Application, Data Files, Platforms and Interfaces) that cannot be delt in the same way. The incremental reengineering process proposed by Olsem uses different strategies for each class of components, reducing the probability of failure during the process.

Another important contribution from Olsem's work is his proposal of two ways to perform incremental reengineering: **with re-integration**, in which the reconstructed modules are reintegrated into the legacy system, and **without re-integration**, in which the modules are identified, isolated and reconstructed, maintaining the interface with the modules that were not submitted to the process through a mechanism called "*Gateway*".

In 2003, Bianchi et al. (Bianchi et al., 2003) presented an iterative model for reengineering aged legacy systems. The proposed process is similar to Olseem's proposed, however, the Bianchi et al. Iterative model focuses on components as result of the reengineering, characteristics are: the reengineering is gradual, i.e., it is iteratively executed on different components (data and functions) in different phases; during the execution of the process, legacy components, components currently undergoing reengineering, reengineered components, and new components, added to the system to satisfy new functional requests, must coexist in harmony.

In another work, Zou and Kontogiannis (Zou & Kontogiannis, 2003) propose an incremental source code transformation framework that allows procedural systems to be migrated to modern object oriented platforms. Initially, the system is parsed and a high level model of the source code is extracted. The framework introduces the concept of a unified domain model for a variety of procedural languages such as *C*, *Pascal*, *COBOL*, and *Fortran*. Next, to keep the complexity and the risk of the migration process into manageable levels, a clustering technique allows the decomposition of large systems into smaller manageable units. A set of source code transformations allow the identification of an object model from each unit. Finally, an incremental merging process allows the amalgamation of the different partial object models into an aggregate composite model for the whole system.

There are several benefits associated with iterative processes: by using the "*divide-and-conquer*" technique, the problem is divided into smaller units, which are easier to manage; the outcomes and return on investment are immediate and concrete; the risks associated to the process are reduced; errors are easier to find and correct, not putting the whole system at risk; and it guarantees that the system will continue to work even during execution of the process, preserving the maintainers' and users' familiarity with the system (Bianchi et al., 2000).

5.3.4 Component-based approaches

Currently, on top of object-oriented techniques, an additional layer of software development, based on components is being established. The goals of "*componentware*" are very similar to those of object-orientation: reuse of software is to be facilitated and thereby increased, software shall become more reliable and less expensive (Lee et al., 2003).

As was discussed in previous Chapters, Component-Based Development (CBD) is not a new idea. McIlroy (McIlroy, 1969) proposed using modular software units in 1968. The extraction of reusable software components from entire systems is an attractive idea, since software objects and their relationships incorporate a large amount of experience from past development. It is necessary to reuse this experience in the production of new software (Caldiera & Basili, 1991).

Among the first research works in this direction, Caldiera and Basili (1991) explore the automated extraction of reusable software components from existing systems. Caldiera and Basili propose a process that is divided in two phases. First, some candidates from the existing system are choosen and packaged for possible independent use. Next, an engineer with knowledge of the application domain analyzes each component to determine the services it can provide. The approach is based on software models and metrics. According to Caldiera and Basili, the first phase can be fully automated, "*reducing the amount of expensive human analysis needed in the second phase by limiting analysis to components that really look worth considering*".

Years later, Neighbors (Neighbors, 1996) presented some informal research, performed over a period of 12 years, from 1980 to 1992, with interviews and the examination of legacy systems, in an attempt to provide an approach for the extraction of reusable components. Although the work does not present conclusive ideas, it gives several important clues regarding large systems. According to Neighbors, the architecture of large systems is a trade-off between top-down functional decomposition and bottom-up support of layers of Application Programming Interfaces (API's) or virtual machines. Therefore, attempts to partition a system according to one of these approaches will not succeed. A better partitioning approach is based on the concept of sub-systems, which are encapsulations convenient to system designers, maintainers and managers. The subsequent step, which may be performed manually or automatically, comprehends their extraction into reusable components.

Another work involving software components and reengineering may be seen in (Alvaro et al., 2003), where Alvaro et al. present a software reengineering CASE environment based on components, called Orion-RE. The environment uses software reengineering and Component-Based techniques to rebuild legacy systems, reusing the available documentation and the built-in knowledge that is in their source code. A software process model drives the environment usage through the reverse engineering, to recover the system design, and forward engineering, where the system is rebuilt using modern technologies, such as design patterns, frameworks, CBD principles and middleware. Alvaro et al. observed some benefits in the reconstructed systems, such as a greater degree of reuse and easier maintenance and also observed benefits due to the automation achieved through the CASE environment.

Other similar approach is proposed in (Lee et al., 2003), where Lee et al. present a process to reengineer an object-oriented legacy system into a component-based system. The components are created based upon the original class relationships that are determined by examining the program source code. The process is composed of two parts: **(i)** creation of basic components with composition and inheritance relationship between constituent classes and **(ii)** refinement of the intermediate component-based system using the Lee et al. proposed metrics, which include connectivity strength, cohesion, and complexity. Finally, the approach proposed by Lee et al. is based on a formal system model, which reduces the possibility of misunderstanding a system and enables operations to be correctly executed.

These four approaches are examples of the current trend on reverse engineering research, as observed by Keller et al. (Keller et al., 1999). Component-Based approaches are being considered in reverse engineering, mainly due to their benefits in reuse and maintainability. However, a complete methodology to reengineer legacy systems into component-based systems is still lacking, but this lack is not restricted to reverse engineering. As can be seen in (Bass et al., 2000), the problems faced when considering Component-Based approaches in reengineering are only a small subset of the problems related to Component-Based Software Engineering in general. While these problems remain unsolved, reengineering may never achieve the benefits associated with software components.

5.3.5 New Research Trends

Figure 5.3 summarizes the survey presented on Section 5.3. In summary, the first works focused on source-to-source translation, without worrying about readability and quality of the generated products. Afterwards, with the appearance of OO technology, there was an increasing concern over the quality of source code and documentation. However, its appearance also introduced problems related to paradigm changing, since most legacy systems were procedural. To reduce these problems, incremental approaches were proposed as alternatives to give more flexibility to the processes, allowing the coexistence of legacy and reengineered systems.

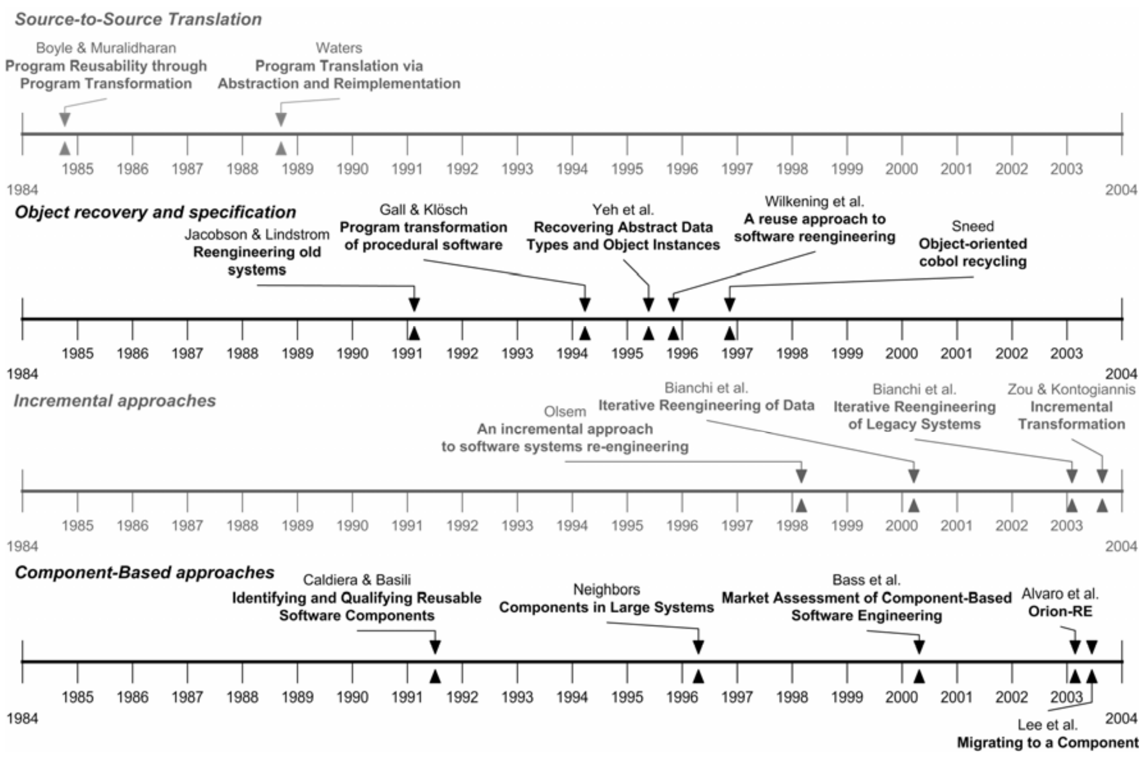


Figure 5.3. Timeline of research on reengineering and reverse engineering.

Component-Based approaches do not follow this evolution (source-to-source → object-orientation → incremental approaches) and have been sparsely researched over the years. This may be explained by the recent nature of CBD and the problems associated with it, inhibiting researchers in their efforts. Althought the isolated efforts never formed a real trend in the reengineering research area, this scenario is currently changing, as it can recently be seen with the concentration of efforts in this direction.

Currently unresolved issues include reducing the functionalities dispersal and increasing modularity, which assist in the maintenance and the evolution of the reengineered systems. Decomposing a system into small parts is a good start to achieve these benefits. Unfortunately, some requirements, such as exception handling or logging, are inherently difficult to decompose and isolate. Object-orientation techniques and Design Patterns (Gamma et al., 1995) may reduce these problems, but they are not enough (Kiczales et al., 1997).

The Aspect-Oriented Software Development (AOSD), came up in the 90's as a paradigm addressed to achieve the "*separation of crosscutting concerns (or aspects)*", through code generation techniques that combine (weave) aspects in the logic of the application (Kiczales et al., 1997). According to aspect-oriented concepts, aspects modify software components through static and dynamic mechanisms. The static mechanisms are concerned with the state and behavior addition in the classes, while the dynamic mechanisms modify the semantics of these at execution time. These aspects are implemented as separate modules, so that it is transparent how aspects act and how they relate to other components (Filman & Friedman, 2000).

Currently, with AOSD technologies being adopted and extended, new challenges and innovations start to appear. AOSD languages, such as AspectJ[1] and AspectS[2], contributions from several research groups[3] and the recent integration with application servers, such as JBOSS[4] and BEA's WebLogic[5], demonstrate the potential of AOSD in solving real problems, including those pursued in reengineering.

Investigations about AOSD in the literature have involved determining the extent to which it can be used to improve software development and maintenance, along the lines discussed by Bayer in (Bayer, 2000). AOSD can be used to reduce code complexity and tangling (intertwined code in a confused mass); it also increases modularity and reuse, which are the main problems that are currently faced by software reengineering. Thus, some works that use AOSD ideas in reengineering may be found in the recent literature.

In Kendall's case study (Kendall, 2000), existing object-oriented models are used as the starting point for reengineering with aspect-oriented techniques. In this work, Kendall did not describe the reengineering process, the techniques, mechanisms nor the following steps in full detail. He just reported the comparative results among the object-oriented code and the aspect-oriented code generated by the reengineering. The use of AOSD in this case study reduced the overall module (number of classes and methods) and Lines Of Code (LOC). There was a reduction of 30 methods and 146 lines of code in the aspect-oriented system.

Sant'anna et al. (Sant'anna et al., 2003) present some metrics for comparisons between aspect-oriented and object-oriented designs and implementations, which may serve to evaluate the product of the reengineering process.

[1] http://eclipse.org/aspectj/
[2] http://www.prakinf.tu-ilmenau.de/~hirsch/Projects/Squeak/AspectS/
[3] http://aosd.net
[4] http://www.jboss.org
[5] http://www.bea.org

In (Lippert & Lopes, 2000), Lippert and Lopes present a study that points the ability of the AOSD in facilitating the separation of the exceptions detection and handling concern. The case study involved the examining and reengineering of a Java-built framework, using AspectJ (Kiczales et al., 2001).

An approach to retrieve the knowledge embedded in an object-oriented legacy system using AOSD is presented in (Garcia et al., 2005). This approach, called the Phoenix approach, aids in the migration from object-oriented code, written in Java, to a combination of objects and aspects, using AspectJ. It uses aspect mining in order to identify possible crosscutting concerns from the OO source code and extracts them through refactorings into new aspect-oriented code. Next, the aspect-oriented design is retrieved through software transformations and may be imported in a CASE tool, and become available in higher abstraction levels. The retrieved information constitutes important knowledge that may be reused in future projects or in reengineering.

An evaluation was conducted to demonstrate the reengineering proccess usefulness. By following Phoenix, it could be verified that AOSD brings several important benefits to software development. The way the aspects are combined with the system modules allows the inclusion of additional responsibilities without committing the clarity of the code, maintainability, reusability, and providing reliability.

In the last two decades, software engineering has focused on application modeling, analysis, simulation, and semi-automated implementation of larger and larger software systems. Recently, the diffusion of personal wireless devices and their highly distributed, heterogeneous, and mobile nature have raised new challenges that permeate the entire software engineering life cycle. Some research involving personal wireless devices has been developed in many directions. This new set of challenges is more appropriately characterized as "*programming in the small and many*" (Medvidovic et al., 2003).

The spread of personal wireless devices has raised the need to port to this new environment existing (legacy) desktop applications. Unfortunately, these applications are often too large. Limited processing power and storage capacity affect the possibility of performing complex computations and working with large datasets, so preventing mobile devices from accessing a wide range of enterprise technologies and resources.

One important work is presented by Canfora et al. (Canfora et al., 2004) called Thin Client aPplications for limiTed dEvices (TCPTE) approach. In this work, Canfora et al. present an approach based on the thin client model implemented through a framework which enables the seamless migration of Java desktop applications to mobile devices with limited resources. The main benefit related to the use of TCPTE is the semi-automatic reengineering of legacy Java AWT applications for their porting to personal wireless devices.

In the last decade, many methods and tools have been proposed to port legacy applications toward new environments through the migration of their user interfaces (UIs) (Canfora et al., 2004). The most known approaches can be classified in: *output analysis, code analysis* and *wrapping, translation*.

These three methods have been primarily used to migrate legacy systems using character based UIs directly onto graphical user interfaces (GUIs) (Merlo et al., 1995), (Aversano et al., 2001) or onto an abstract description for a successive implementation on GUIs, WEB or WAP UIs (Kapoor & Strolia, 2001), or to migrate GUIs from one toolkit to a new one (Moore & Moshkina, 2000).

With the translation approach (Paulson, 2001) the legacy application code is transformed in order to employ the primitives supported by the target platform.

Finally, with the library substitution approach, the legacy application code does not need to be modified because the GUI framework is replaced at linking time.

Far too often, architecture descriptions of existing software systems are out of synchronization with the implementation. If they are, they must be reconstructed, but this is a very challenging task.

5.4 Towards an effective approach for Reverse Engineering

The survey presented in Section 5.3 reveals that research into reverse engineering covers many different issues. Each one of the four research trends has its own focus, giving its own contribution to the area. These different issues served as a basis for the definition of a set of requirements for an effective reverse engineering approach, which will be presented in next sections.

5.4.1 Generic process

At some level of detail, reverse engineering projects will differ in terms of their processes. Different tasks are needed to achieve different types of changes. A process model created to guide a specific reverse engineering project must be based on characteristics of the system to be reengineered and the planned changes. The collection and sequence of tasks that occur during a project, the structure and semantics of project information, the tools used, and the human roles and relationships that make up the organizational structure of a project are all specific to that project. But the basic process of planning and executing a reverse engineering project should be the same in all projects.

5.4.2 Define objectives first

All changes in system implementation should be based on strategic objectives, expressed in a strategic plan and later implemented (Bianchi et al., 2003).

According to Olsem (1998) the objectives can be divided in three categories: organizational, system and project. Each category refers to a different group of concerns about the project and what should be reached.

Organizational objectives specify long-term and short-term goals, legacy system problems addressed by the project, procedures that are to be followed, standards to which project components must adhere, tools that are to be used, among others.

System objectives typically specify changes to the implementation, design or requirements of a system. System objectives tend to contain action words, such as "*change*", "*add*" or "*replace*".

Project objectives focus on concerns such as the project duration, staffing details and particular methods to be used. Specific deliverables of the reverse engineering project should be stated up front. A project objective could be "*to modularize the system through aspect-orientation*", for example.

5.4.3 Form a reverse engineering team

Within every project there must be an organizational structure (Olsem, 1998) to assign project personnel to project roles. Each project member fills one or possibly several roles within a project. These roles include project manager, team leader, coordinator, programmer, database management system administrator or tester, amongst others. Each role has a title level of authority and requires a set of qualifications.

The team will also include adjunct members, such as legacy system specialists and support staff, with knowledge on the technologies involved in the reverse engineering process.

5.4.4 Define metrics

Metrics constitute a baseline to measure the results of the reverse engineering project. They should directly correspond to the strategic goals and objectives, which were defined earlier. These metrics should also include a means to measure the project status/progress during each phase.

With the appearance of AOSD, metrics are being defined to evaluate how better or worse is the aspect-oriented system in comparison with your object-oriented version (Sant'anna, 2003).

5.4.5 Analyze all possible information

In order to capture the information that will be used in the reverse engineering, it is necessary to perform a full analysis of the legacy system's elements. There is a variety of elements to be collected, including: programs, library routines, business rules, informal documentation, models, user manuals and history records, amongst others. These constitute the knowledge to be reused.

The software engineer starts from legacy code and all the available information to identify the legacy system components. In the case of object-oriented reverse engineering (Gall & Klösch, 1994), (Sneed, 1996) and (Keller et al., 1999), the software engineer can identify in the legacy code the elements that will compose the object-oriented models of the legacy system.

Relationships among components must also be considered and stored as attributes of the related components. One such relationship is the "*depends on*" attribute which indicates that one component depends on another component. For example, a document that describes module interfaces in a program depends on that program and also depends on the specific modules documented. If the modules are changed, the documentation might have to be changed as well.

5.4.6 Organize the legacy system

Since the reengineering product must be prepared for future maintenance and reuse, an intermediate step, which organizes the recovered information in order to achieve larger modularization, is required. This facilitates the task of recovering the documentation and reconstructing the system.

Techniques can be used to aid the software engineer in restructuring the source code. One of such techniques is refactoring. Refactoring can be defined as "*a set of well planned techniques with the objective of improving the capacity of the system to adapt to new needs and to accommodate new functionalities*" (Fowler et al., 1999).

5.4.7 Reconstruct documentation

For every system to be developed/evolved, there is a tradeoff between having detailed documentation, which will be expensive to maintain, and having little documentation, which will make maintenance harder.

In reverse engineering, documentation requires additional flexibility due to the dynamic nature of succeeding incremental projects, parallel development efforts on different components, and the need to smoothly integrate the combined documentation of lower level components to represent higher levels of abstraction.

5.4.8 *"Divide–and-conquer"*

The reconstruction of the whole system in a single step brings many problems, such as delay on results, return on investment, higher risks and difficulty to finding and correcting errors. Incremental approaches have proven to be the best way to reduce these problems. Dividing the legacy system into smaller subsystems has been proposed by several works, such as (Olsem, 1998), (Bianchi et al., 2000) and (Zou & Kontogiannis, 2003).

However, to correctly divide legacy systems is not easy. The interdependency between modules along with other aspects, such as modules' importance and relevance, which are specific to each project, may also influence the decision process. Some works offer some clues in order to facilitate this task. Neighbors' (Neighbors, 1996) discusses some strategies to perform such division, and Olsem's (1998) proposes the separation of the modules according to four classes of components.

Finally, just dividing the system is not enough. There must be coexistence between the legacy modules, modules being reconstructed and new modules. Wrapping techniques (Bianchi et al., 2000) are a good way to manage this coexistence.

5.4.9 Automated support

Automation tools offer valuable help in the reengineering process, saving time, effort and costs. However, tools must be chosen based on the project's strategic goals/objectives and available resources (funding, personnel and time). Commonly, tools are chosen before these plans are defined, which may lead to wrong tools adoption and unnecessary costs. In order to avoid this difficulty, a reengineering tool suite that fulfills the organizational, system and project needs outlined by the first step of this process must be acquired (Olsem, 1998).

There are different classes of tools, which may act on different abstraction levels. There are parsers or translators (Boyle & Muralidharan, 1984), (Waters, 1988), transformational systems (Gall & Klösch, 1994), (Zou & Kontogiannis, 2003), code analysis tools (Wilkening et al., 1995), object extraction tools (Sneed, 1996) and modeling tools (Alvaro et al., 2003), (Jacobson & Lindstrom, 1991).

5.5 Chapter Summary

Many reverse engineering and reengineering approaches have been proposed to derive abstract representations from legacy systems. The goal is to develop a global picture of the subject system, which is the first major step towards its understanding or transformation into a system that better reflects the quality needs of the application domain.

This chapter presented a survey that identified four research trends of the reverse engineering and reengineering fields, from the late 80's to nowadays: **(i)** source-to-source translation, **(ii)** object recovery and specification, **(iii)** incremental approaches and **(iv)** component-based approaches. Researches will continue to develop technology and tools for generic reverse engineering tasks, but future research ought to focus on ways to make the process of reverse engineering more repeatable, defined, managed, and optimized (Müller et al., 2000).

The most promising direction in this area is the continuous program understanding approach. The premise is that reverse engineering needs to be applied continuously throughout the lifetime of the software in order to understand and potentially reconstruct the earliest design and architectural decisions of the system. Tool integration and adoption should be central issues. Organizations should understand this premise and obtain full managerial support in order to accomplish these goals.

For the future, the authors agree with (Müller et al., 2000), it is critical that the software engineers effectively answer questions, such as "*How much knowledge, at what level of abstraction, do we need to extract from a subject system, to make informed decisions about reengineering it?*". Thus, there is a need to tailor and adapt the program understanding tasks to specific reengineering objectives.

A new tendency is that the research moves towards the separation of concerns area, based on AOSD concepts, integrated with already existing reverse engineering and reengineering techniques. This represents one step further in the post object-orienation era, towards even higher maintainability and flexibility levels.

Next Chapter presents the relevance of reuse metrics and describes the most common metrics defined in the literature to measure software reuse.

5.6 References

(Alvaro et al., 2003) Alvaro, A.; Lucrédio, D.; Garcia, V. C.; Almeida, E. S.; Prado, A. F.; Trevelin, L. C. **Orion-RE: A Component-Based Software Reengineering Environment**. In: *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE)*, pp. 248–257. IEEE Computer Society Press, November 2003.

(Aversano et al., 2001) Aversano, L.; Cimitile, A.; Canfora,; De Lucia, A. **Migrating Legacy Systems application to the Web.** In: *Proceedings of 5th European Conference on Software Maintenance and Reengineering*, Lisbon, Portugal, IEEE Comp. Soc. Press, 2001, pp.148-157.

(Bass et al., 2000) Bass, L.; Buhman, C.; Comella-Dorda, S.; Long, F.; Robert, J.; Seacord, R.; Wallnau, K. **Volume i: Market assessment of component-based software engineering**. In: *Technical note CMU/SEI-2001-TN-007, Carnegie Mellon University, Software Engineering Institute*, May 2000.

(Bayer, 2000) Bayer, J. **Towards engineering product lines using concerns**. In: *Workshop on Multi-Dimensional Separation of Concerns in Software Engineering (ICSE)*, June, 2000.

(Bennett & Rajlich, 2000) Bennett, K. H.; Rajlich, V. T. **Software maintenance and evolution: a roadmap**. In: *Proceedings of the 22nd International Conference on Software Engineering (ICSE). Future of Software Engineering Track*, pp. 73–87. ACM Press, 2000.

(Bianchi et al., 2003) Bianchi, A.; Caivano, D.; Marengo, V.; Visaggio, G. **Iterative reengineering of legacy systems**. In: *IEEE Transaction on Software Engineering,* Vol. 29, No. 03, March, pp. 225–241, 2003.

(Bianchi et al., 2000) Bianchi, A.; Caivano, D.; Visaggio, G. **Method and process for iterative reengineering of data in a legacy system**. In: *Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE)*, pp. 86–97. IEEE Computer Society, 2000.

(Biggerstaff et al., 1994) Biggerstaff, T. J.; Mitbander, B. G.; Webster, D. E. **Program understanding and the concept assignment problem**. In: *Proceedings of the 15nd International Conference on Software Engineering (ICSE)*, pp. 482-498. ACM Press, 1993.

(Boyle & Muralidharan, 1984) Boyle, J. M.; Muralidharan, M. N. **Program reusability through program transformation**. In: *IEEE Transactions on Software Engineering*, *Special Issue on Software Reusability*, Vol. 10, No. 5, pp. 574–588, September, 1984.

(Caldiera & Basili, 1991) Caldiera, G.; Basili, V. R. **Identifying and qualifying reusable software components**. In: *IEEE Computer*, Vol. 24, No. 02, pp. 61–71, February, 1991.

(Canfora et al., 2004) Canfora, G.; Di Santo, G.; Zimeo, E. **Toward Seamless Migration of Java AWT-Based Applications to Personal Wireless Devices.** In: *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE)*, pp. 38-47, 2004.

(Chidamber & Kemerer, 1999) Chidamber, S. R.; Kemerer, C. F. **A metrics suite for object oriented design**. In: *IEEE Transaction on Software Engineering*, Vol. 20, No. 06, June, 1994, pp. 476–493.

(Chikofsky & Cross, 1990) Chikofsky, E. J.; Cross, J. H. **Reverse engineering and design recovery: a Taxonomy**. In: *IEEE Software*, Vol. 01, No. 07, January, 1990, pp. 13–17.

(Dickinson, 1997) Dickinson, I. J. **Agents standards**. In: *Technical Report HPL-97-156, Hewlett Packard Laboratories*, December, 1997.

(Favre, 2004) Favre, J.M. **CacOphoNy: Metamodel-Driven Architecture Recovery**. In: *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE)*, pp. 204-213, 2004.

(Filman & Friedman, 2000) Filman, R. E.; Friedman, D. P. **Aspect-oriented programming is quantification and obliviousness**. In: *Workshop on Advanced Separation of Concerns (OOPSLA)*, October, 2000.

(Fowler et al., 1999) Fowler, M.; Beck, K.; Brant, J.; Opdyke, W.; Roberts, D. **Refactoring: improving the design of existing code**. *Object Technology Series. Addison-Wesley*, 1999.

(Gall & Klösch, 1994) Gall, H.; Klösch, R. **Program transformation to enhance the reuse potential of procedural software.** In: *Proceeding of the ACM Symposium on Applied Computing (SAC4)*, pp. 99–104. ACM Press, 1994.

(Gamma et al., 1995) Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. **Design Patterns - Elements of Reusable Object-Oriented Software**. *Addison Wesley Professional Computing Series*. Addison-Wesley, 1995.

(Garcia et al., 2005) Garcia, V. C.; Lucrédio, D.; Prado, A. F.; Almeida, E. S.; Alvaro, A.; Meira, S. R. L. **Towards an Approach for Aspect-Oriented Software Reengineering**. In: *7th International Conference on Enterprise Information Systems (ICEIS)*, Miami, USA, May, 2005.

(Jacobson & Lindstrom, 1991) Jacobson, I.; Lindstrom, F. **Reengineering of old systems to an object-oriented architecture**. In: *Proceedings of the Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'91)*, pp. 340–350. ACM Press, 1991.

(Kapoor & Stroulia, 2001) Kapoor, R. V.; Stroulia, E. **Mathaino: Simultaneous Legacy Interface Migration to Multiple Platforms.** In: *9th International Conference on Human-Computer Interaction*, August 5-10, 2001, New Orleans, LA, USA, Vol. 01, pp. 51-55, Lawrence Erlbaum Associates.

(Keller et al., 1999) Keller, R. K.; Schauer, R.; Robitaille, S.; Pagé, P. **Pattern-based reverse engineering of design components**. In: *Proceedings of the 21st International Conference on Software Engineering (ICSE)*, pp. 226–235. IEEE Computer Society Press, 1999.

(Kendall, 2000) Kendall, E. A. **Reengineering for separation of concerns**. In: *Workshop on Multi-Dimensional Separation of Concerns in Software Engineering (ICSE)*, June, 2000.

(Kiczales et al., 2001) Kiczales, G.; Hilsdale, E.; Hugunin, J.; Kersten, M.; Palm, J.; Griswold, W. G. **Getting started with AspectJ**. In: *Communications of ACM*, Vol. 44, No. 10, October, 2001, pp. 59–65.

(Kiczales et al., 1997) Kiczales, G.; Lamping, J.; Mendhekar, A.; Maeda, C.; Lopes, C.; Loingtier, J.M.; Irwin, J. **Aspect-Oriented Programming**. In: *Proceedings of the 11st European Conference Object-Oriented Programming (ECOOP)*, Vol. 1241, pp. 220–242. Springer Verlag, 1997.

(Lee et al., 2003) Lee, E.; Lee, B.; Shin, W.; Wu, C. **A reengineering process for migrating from an object-oriented legacy system to a component-based system**. In: *Proceedings of the 27th Annual International Computer Software and Applications Conference (COMPSAC)*, pp. 336–341. IEEE Computer Society Press, November 2003.

(Lehman & Belady, 1985) Lehman, M. M.; Belady, L. A. **Program Evolution: Processes of Software Change**, Vol. 27 of *APIC Studies in Data Processing*. Academic Press, 1985.

(Lippert & Lopes) Lippert, M.; Lopes, C. V. **A study on exception detecton and handling using aspect-oriented programming**. In: *Proceedings of the 22nd International Conference on Software Engineering (ICSE)*, pp. 418–427. ACM Press, 2000.

(McIlroy, 1969) McIlroy, M. D. **Mass produced software components**. In: *Software Engineering: Report on a conference sponsored by the NATO Science Committee*, pp. 138–155. NATO Scientific Affairs Division, January. 1969.

(Medvidovic et al., 2003) Medvidovic, N.; Rakic, M. M.; Mehta, N.; Malek, S. **Software Architectural Support for Handheld Computing.** In: IEEE *Computer*, Vol. 36, No. 09, September, 2003, pp.66-73.

(Merlo et al., 1995) Merlo, E.; Gagné, P.Y.; Girard, J.F.; Kontogiannis, K.; Hendren, L.; Panangaden, P.; De Mori, R. **Reengineering User Interfaces.** In: *IEEE Software*, Vol. 12, January/February, 1995, pp. 64-73.

(Meyer, 1997) Meyer, B. **Object-Oriented Software Construction**. *Prentice-Hall, Englewood Cliffs*, second edition, 1997.

(Moore & Moshkina, 2000) Moore, M. M.; Moshkina, L. **Migrating Legacy User Interfaces to the Internet: Shifting Dialogue Initiative**. In: *Proceedings of 7th Working Conference on Reverse Engineering (WCRE)*, Brisbane, Australia, IEEE Comp. Soc. Press 2000, pp. 52-58.

(Muller et al., 2000) Müller, H. A.; Jahnke, J. H.; Smith, D. B.; Storey, M. A.; Tilley, S. R.; Wong, K. **Reverse engineering: a roadmap**. In: *Proceedings of the 22nd International Conference on Software Engineering (ICSE)*. Future of Software Engineering Track, pp. 47–60. ACM Press, 2000.

(Neighbors, 1996) Neighbors, J. M. **Finding reusable software components in large systems**. In: *Proceedings of the 3rd Working Conference on Reverse Engineering (WCRE)*, pp. 02-10. IEEE Computer Society, 1996.

(Olsem, 1998) Olsem, M. R. **An incremental approach to software systems re-engineering**. In: *Journal of Software Maintenance*, Vol. 10, No. 03, May/June, 1998, pp. 181-202.

(Ossher & Tarr, 1999) Ossher, H.; Tarr, P. **Using subject-oriented programming to overcome common problems in object-oriented software development/evolution**. In: *Proceedings of 21st International Conference on Software Engineering (ICSE)*, pp. 687–688. IEEE Computer Society Press, 1999.

(Paulson, 2001) Paulson, L. D. **Translation Technology Tries to Hurdle the Language Barrier**. In: *IEEE Software*, Vol. 34, No. 9, September, 2001, pp.12-15.

(Sant'anna et al., 2003) Sant'anna, C.; Garcia, A.; Chavez, C.; von Staa, A.; Lucena, C. **On the reuse and maintenance of aspect-oriented software: An evaluation framework**. In: *XVII Brasilian Symposium on Software Engineering*, October 2003.

(Sneed, 1995) Sneed, H. M. **Planning the reengineering of legacy systems**. In: *IEEE Software*, Vol. 12, No. 01, January, 1995, pp. 24–34.

(Sneed, 1996) Sneed, H. M. **Object-oriented cobol recycling**. In: *Proceedings of the 3rdWorking Conference on Reverse Engineering (WCRE)*, pp. 169–178. IEEE Computer Society Press, January 1996.

(Sommerville, 1996) Sommerville, I. **Software Engineering**. *Addison-Wesley*, fifth edition, 1996.

(Stevens & Pooley, 1998) Stevens, P.; Pooley, R. **Systems reengineering patterns**. In: *Proceedings of the ACM SIGSOFT 6th International Symposium on the Foundations of Software Engineering (FSE)*, Vol. 23, No. 06, Software Engineering Notes, pp. 17–23, November, 1998.

(Systa, 1999) Systa, T. **The relationships between static and dynamic models in reverse engineering java software**. In: *Proceedings of the 6th Working Conference on Reverse Engineering (WCRE)*. IEEE Computer Society Press, October 1999.

(Umar, 1997) Umar, A. **Application (Re)Engineering: Building Web-Based Applications and Dealing with Legacies**. *Prentice-Hall*, Upper Saddle River, NJ, 1997.

(Visaggio, 2001) Visaggio, G. **Ageing of a data intensive legacy system: Symptoms and remedies**. In: *Journal of Software Maintenance and Evolution*, Vol. 13. No. 05, September/October, 2001, pp. 281–308.

(Waters, 1988) Waters, R. C. **Program translation via abstraction and reimplementation**. In: *IEEE Transaction on Software Engineering*, Vol. 14, No. 08, August, 1988, pp. 1207-228.

(Wilkening et al., 1995) Wilkening, D. E.; Loyall, J. P.; Pitarys, M. J.; Littlejohn, K. **A reuse approach to software reengineering**. In: *Journal of Systems and Software*, Vol. 30, No. 01-02, July/August, 1995, pp. 117–125.

(Yeh et al., 1995) Yeh, A. S.; Harris, D. R.; Reubenstein, R. **Recovering abstract data types and object instances from a conventional procedural language**. In: *Proceedings of the Second Working Conference on Reverse Engineering (WCRE)*, pp. 227–236. IEEE Computer Society Press, 1995.

(Zou & Kontogiannis, 2003) Zou, Y.; Kontogiannis, K. **Incremental transformation of procedural systems to object oriented platforms**. In: *Proceedings of the 27th Annual International Computer Software and Applications Conference (COMPSAC)*, pp. 290–295. IEEE Computer Society Press, November, 2003.

Chapter 6
Software Reuse Metrics

Since the time that the concept of software reuse was first discussed, many intriguing challenges have been posed to the software engineering community: if on the one hand it is common sense that software reuse can potentially have a positive impact on the outcome of a software production process, on the other hand there is a general belief that taking actual software reuse to a level in which it becomes economically relevant is no easy task and for that it should be confined to the walls of specific domains.

The fundamental questions in this context are then: "how much software reuse?" and "how much better with software reuse?". These questions lead us to another important software engineering area closely related to software reuse: software metrics. The universally accepted truth that what cannot be measured, cannot be managed also holds for the software engineering field and particularly to the area of software reuse.

Existing software reuse metrics are divided into two main categories (Mascena et al., 2005): *Economics Oriented Reuse Metrics* and *Models* (EORM), and software *Structure Oriented Reuse Metrics* (SORM). Economics oriented metrics and models aim to assess the impacts of reuse programs in organizations and are associated with return on investment (ROI) models while software structure metrics are generally concerned with what is being reused and how it is being reused from a technical standpoint.

Another software reuse metrics category, *Reuse Repository Metrics* (RRM), is related to reuse repositories that target the assessment of reuse repository aspects such as availability of the repository, search engine performance, quality of available assets and number of times assets were successfully reused. Such metrics can be used as support tools for determining where organizations should direct their effort regarding maintenance and evolution of reusable assets available in the repository and potential new reusable assets to be developed.

Figure 6.1. shows how the reuse metrics field is divided. The remainder of this chapter will be devoted to describing the state-of-the-art of software reuse metrics in the three main categories.

Reuse Metrics

- **Economics Oriented Reuse Metrics (EORM)** are related to return on investment (ROI) models and the economical impacts of software reuse based on a set of observable data;
- **Software Structure Oriented Reuse Metrics (SORM)** are concerned on what is being reused and how it is being reused from a strictly technical standpoint;
- **Reuse Repository Metrics (RRM)** assess the overall performance of the convergence point of the reuse activities in an organization.

Figure 6.1. Main reuse metrics categories.

6.1 Economics Oriented Metrics

If you think about it, engineering is all about money. Every decision an engineer makes must take into account the economic impacts related to that decision and metrics are important instruments for assessing such impacts.

Economics oriented reuse metrics are mainly concerned with the economical aspects related to reuse programs in organizations and are the basic instruments for organization-wide return on investment models (ROI).

Cost and productivity models measure the cost of reusing software components and the cost of developing reusable components (Barnes, 1988) (Gaffney & Durek, 1989). The reuse benefit corresponds to how much was saved by reusing existing reusable components. The ratio of reuse benefits to reuse investments determines if the reuse effort resulted in profit or loss to the organization (Barnes & Bollinger, 1991).

Economics oriented metrics are based on a set of observable data (Poulin & Caruso, 1993), (Poulin, 2002): *Shipped Source Instructions* (SSI), *Changed Source*

Instructions (CSI), *Reused Source Instructions* (RSI), *Source Instructions Reused by Others* (SIRBO) and data related to software development cost and quality, such as software development cost (Cost per LOC), software development error rate (Error rate) and software error repair cost (Cost per error).

From this set of observable data, the most relevant one is **SIRBO** which accounts for the amount of code contributed by a team or organization to a reuse program is being actually reused by other teams or organizations. This is very important information for any reuse incentive program or reuse repository metric as discussed in Section 6.3.

Besides the observable data, two additional abstractions must be presented before defining the main economics oriented reuse metrics: Relative Cost of Reuse (RCR) and Relative Cost of Writing Reusable Software (RCWR).

Relative Cost of Reuse (RCR) is the cost of activities associated with finding and integrating reusable assets into newly developed applications. This cost is intrinsically related to a reuse educational program and adequate tool support to the developers, as well as the quality of existing reusable assets, which in turn depends on the reuse education that developers are exposed to. Reuse initiatives will be severely inhibited if the overall RCR is high, so it is essential to have all these prerequisites well established right at the beginning of any reuse program.

At this point we face another dilemma: the cost for having all these prerequisites in place right at the beginning of a reuse program, when there are no guaranties that this investment will have a proper return in a reasonable time-frame, might be a huge inhibitor for the adoption of such a program. There is no easy answer to this problem, but an incremental model in which each level has a set of requirements and an expected return might be part of the answer.

Relative Cost of Writing Reusable Software (RCWR) is the additional cost of writing software concerned with reusability issues. Such concerns include generalization of the asset to meet additional requirements instead of just meeting the immediate requirements, manufacturing more detailed documentation so developers reusing that asset are able to easily understand how it works and writing and executing test code to increase trust in the assets to be reused.

Once the observable data and the basic abstractions are established, the economic value of reuse may be directly derived into the main economics oriented metrics: *Reuse Cost Avoidance* (RCA), *Organizational or Project-level ROI* and *Reuse Value Added* (RVA). Table 6.1 summarizes the definitions of the main economics related reuse metrics.

Table 6.1. Main economics related software reuse metrics.

Metric	Definition
Reuse Cost Avoidance (RCA)	The cost saved by reusing software instead of writing new software from scratch.
Reuse Value Added (RVA)	The reuse effectiveness of a team or organization based on what is being reused by the team (RSI), what from the team is being reused by others (SIRBO) and the total size of the system (SSI).
Organizational or Project-level ROI	The difference between the costs saved by reuse (RCA) and the additional costs for obtaining reuse.

Reuse Cost Avoidance (RCA) is the cost saved by writing code with reusable assets compared to an estimation of the cost of writing an equivalent code without reusable assets. This metric takes into account not only the development phase, which is usually the case for other defined metrics in the literature, but also the maintenance or service phase. The rationale is that by reusing assets, less code must be produced, thus reducing overall development effort, combined with the fact that a reusable asset has potentially fewer errors than a newly written asset, since it has been tested in previous products using that same asset. Fewer errors mean less time fixing errors and consequently less overall effort during development and maintenance. Suppose the cost for writing and maintaining an application throughout its entire lifecycle is estimated at \$40K and by reusing available assets that cost drops to \$32K, the RCA is \$8K. The calculation of RCA is based on RCR.

Reuse Value Added (RVA) measures the effectiveness a team or an organization has in a reuse initiative program. It is calculated by adding the total size of the shipped software by a team or organization (SSI) to what the team or organization is reusing from the reuse library (RSI) and what was produced by that team or organization and is being reused by others (SIRBO). The result of this sum is then divided by SSI. The resulting index is a measure of the reuse effectiveness of that team or organization: an RVA=1 means there is no reuse activity involved at all, since

RSI+SIRBO=0, while an RVA=2, for instance, means that the team or organization is twice as effective from a reuse perspective.

This was one of the first defined metrics, but since people usually do not find indexes very intuitive it has been replaced by the more economics oriented Organizational or Project-level ROI metric.

Organizational or Project-level ROI is calculated subtracting the additional development effort of the organization or the project, depending on the level this metric is being calculated, from the previously defined RCA. This is the most complete economically oriented metric since it takes into account all the costs and benefits involved with a reuse initiative program. In order to have accurate estimates of this metric, though, there must be a well defined development process along with appropriate tool support to keep track of all the reuse related activities.

6.1.1 Software Reuse Impacts

Empirical studies, in both industry and academia, with the aim of assessing the relationship between software reuse and different quality and cost metrics have been reported in the literature (Lim, 1994), (Henry and Faller, 1995), (Basili et al., 1996), (Devanbu et al., 1996) and (Frakes and Succi, 2001). All of the reported studies dealt with a very limited number of projects, which made their results inconclusive, but the general notion that software reuse and software quality are intrinsically related held true for all cases, while the inverse relation between software reuse and development cost failed to hold for some of the studies. Table 6.2 summarizes the measurable impacts of software reuse.

Table 6.2. Measurable impacts of software reuse.

Aspect	Measurable Impacts
Quality	• Error density • Fault density • Ratio of major errors to total faults • Rework effort • Module deltas • Developers perception
Productivity	• Lines of code per effort
Time-to-Market	• Development cycle time

Error density is the average number of severe errors a piece of software presents per line of code, while **fault density** accounts for less severe errors. The studies show that projects with higher reuse activity tend to have lower error density. The reason is that a reused piece of software has been tested and debugged in previous systems, thus leading to fewer errors. Besides having fewer errors, the **ratio between major errors and total number of faults** tends to be smaller for projects that reuse more software.

As direct consequences, the overall **rework effort** and the number of **module deltas** tend to be smaller. Since there are fewer errors, less effort must be spent fixing errors and fewer changes (deltas) will be necessary. The **software quality as perceived by developers** is a subjective measure based on the experience of the developers during the development process. Developers fill out forms describing their impressions of the quality of the software built and the difficulties they had to deal with and the results are compared between projects that considered reuse and projects that did not consider reuse during the entire development cycle.

Although there is no definitive conclusion about the actual impacts software reuse has on different aspects such as quality and cost, studies have shown that there is a correspondence between them.

6.2 Software Structure Oriented Metrics

The whole point of software reuse is achieving the same or better results at the same or smaller cost when compared to a non-reuse oriented software development approach. From this perspective, the previous sections on economically oriented metrics and software reuse impacts would be enough for the reuse metrics field. The problem with these metrics is that they rely on a set of basic observable data that in some cases may lead to incorrect results.

Such metrics are concerned on *how much* was reused versus how much was developed from scratch, but fail to help on the analysis of *what* was reused and *how* it was reused. Software structure oriented metrics aim to fill this gap by providing more elaborate ways of analyzing the relationship between reused and new code on a software system.

The software structure oriented metrics are divided into two main categories: the **amount of reuse metrics** and the **reusability assessment metrics**. The former target assessing the reuse of existing items, while the later aim to assess, based on a set of quality attributes, how reusable items are. Table 6.3 summarizes the main amount of reuse metrics defined so far.

Table 6.3. Amount of reuse metrics.

Metric	Definition
Reuse Percent (RP)	Ratio of the number of reused lines of code to the total number of lines of code (Poulin and Caruso, 1993).
Reuse Level (RL)	Ratio of the number of reused items to the total number of items (Frakes and Terry, 1994).
Reuse Frequency (RF)	Ratio of the references to reused items to the total number of references (Frakes and Terry, 1994).
Reuse Size and Frequency (RSF)	Similar to Reuse Frequency, but also considers the size of the items in number of lines of code (Devanbu et al., 1996).
Reuse Ratio (RR)	Similar to Reuse Percent, but also considers partially changed items as reused (Devanbu et al., 1996).
Reuse Density (RD)	Ratio of the number of reused parts to the total number of lines of code (Curry et al., 1999).

For the explanation of the different amount of reuse metrics, consider the example application (**SampleApp**) with the structure depicted in Figure 6.2. Each part (square) corresponds to an item (e.g. a file) that has a name and a size in number of lines of code (LOC) and the arrows correspond to references between the items (e.g. method calls). Solid arrows are references to internal items and dashed arrows are references to external items. The application is composed of internal, or new parts (1, 2, 3 and 4), the white squares, and external, or reused, parts (5 and 6), the gray squares.

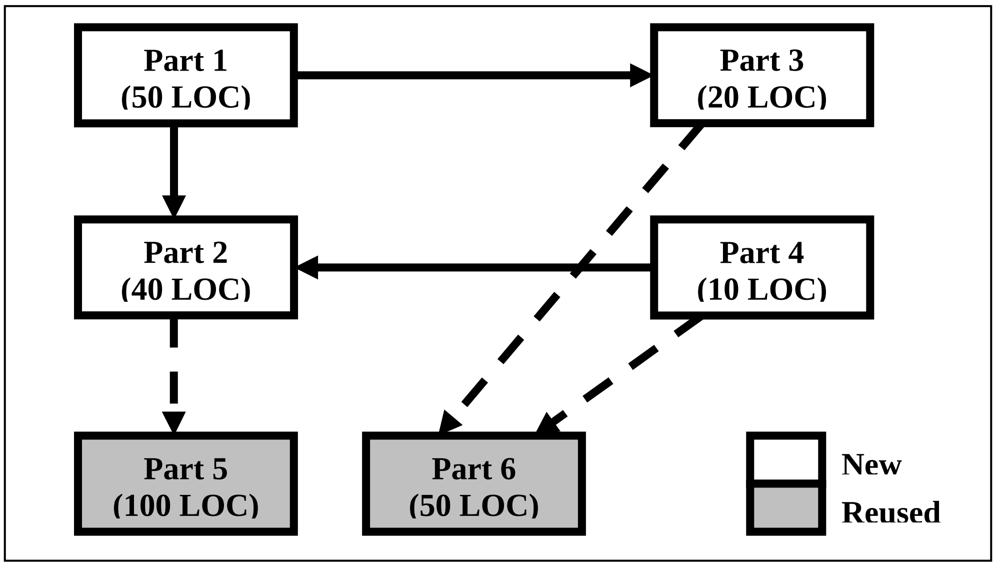


Figure 6.2. Sample application (SampleApp) structure.

Reuse Percent (RP) is the most basic reuse metric, used in current economics oriented reuse metrics. It is defined as the ratio of the number of reused lines of code to the total number of lines of code. The RP for SampleApp is the calculated as follows:

```
Σ LOC(reused) = LOC(part 5) + LOC(part 6) = 150
Σ LOC(new) = Σ LOC(parts 1 through 4) = 120
RP(SampleApp) = (150 / (150 + 120)) * 100 = 55.6%
```

Although this is a very simple metric to understand and extract from existing software systems, it can lead to incorrect conclusions. If part 5, for example, had 2000 lines of code instead of 100, the RP would be totally different from the previously calculated (94.47%), even if SampleApp reused exactly the same set of functionalities from part 5.

Reuse Level (RL) is the ratio of the number of reused parts to the total number of parts. Internal parts may also be considered as reused, when there is more than a reference to it. In this case, RL may be divided into Internal RL (IRL), external RL (ERL) and Total RL (TRL). For SampleApp, the RL, which is equivalent to ERL, is calculated as:

```
Count(reused parts) = Count(part 5, part 6) = 2
Count(all parts) = Count(parts 1 through 6) = 6
RL(SampleApp) = 2 / 6 = 0.33
```

This is also a very simple metric to understand and extract but it has two main problems: first, large systems are penalized because the number of reused parts tend to stabilize while the number of new parts grow with the size of the system, and second, poorly designed systems may benefit from the fact of having fewer new parts compromising modularity (spaghetti, monolithic code).

Reuse Frequency (RF) is defined as the ratio of the number of references to reused parts to the total number of references. Similarly to RL, RF may be divided into Internal RF (IRF), External RF (ERF) and Total RF (TRF). For SampleApp, the RF is:

```
Refs(reused parts) = Refs(parts 5 and 6) = 3
Refs(all parts) = Refs(parts 1 through 6) = 6
```
RF(SampleApp) `= 3 / 6 =` **0.5**

This metric has a similar problem compared to RL: poorly designed systems may incorrectly benefit from the fact that there are few internal references because the vast majority of the functionalities are located at few parts.

Reuse Size and Frequency (RSF) measures reuse considering the size of the parts and the number of references to the reused parts. It can be seen as a combination of RP and RF. For this metric, the concept of expanded size of the system, which consists of the size of the system considering the size of the parts for every reference to those parts, is introduced. It is similar to macro expansion schemas existent on some programming languages: every call to a macro is replaced by the actual macro code (inline macro expansion). For SampleApp, RSF is calculated as follows:

```
Size = Σ LOC(new parts) = LOC(parts 1 through 4) = 120
Expanded Size = Σ LOC(part) * Refs(part) =
50*1 + 40*2 + 20*1 + 10*1 + 100*1 + 50*2 = 360
```
RSF (SampleApp) `= (Expanded Size – Size) / Expanded Size =` **0.67**

The rationale behind this metric is that if the system had no reuse at all, every time a part is referenced, the code equivalent to that part would have to be written again. The problem with this metric is the same with RP: large reused parts may incorrectly interfere in the result leading to wrong conclusions.

Reuse Ratio (RR) is similar to RP, but besides black-box (verbatim) reuse, it also considers white-box reuse: items changed to a certain degree are also considered as reused. For SampleApp, considering that all reused parts are reused verbatim, RR is equal to RP.

Reuse Density (RD) is the ratio of the number of reused parts to the size of the system in number of lines of code. As in RF and RL, this metric may be divided into

Internal RD (IRD), External RD (ERD) and Total RD (TRD). For SampleApp, RD is calculated as follows:

`RD (SampleApp)` `= Count(reused parts) / Σ LOC(all parts) = 2 / 270 =` **`0.007`**

This metric also penalizes large systems because it does not consider how the parts are being reused. As the size of the system grows the number of reused parts tend to stabilize, resulting in smaller RD values. Had SampleApp two more parts that reused parts 5 and 6, for example, the RD would be smaller.

It is important to stress that no single metric is able to depict precisely how a system or organization is in terms of reuse and perfectly reflect its impacts. A judicious use of a combination of metrics and observations is needed in order to accurately estimate the actual consequences of a software reuse program. Table 6.4 summarizes the metrics values for SampleApp. From this table, it is clear that each metric represents certain aspects of the application. A thorough analysis of such metrics is necessary for a better understanding of the reuse reality of the application, taking the context of the application into account.

Table 6.4. SampleApp reuse metrics values.

Metric	Value
RP	55.6%
RL	0.33
RF	0.5
RSF	0.67
RR	0.56
RD	0.007

The RP and RL values, for example, indicate that the application is reusing parts proportionally larger than the new parts being built. Depending on the context of the application and on the parts that are being reused, this may be a good or bad thing. The bottom line is: there is no general rule of thumb for analyzing amount of reuse metrics.

Instead of assessing how much was reused from previously existing reusable parts when building new applications, **reusability assessment metrics** aim to assess how reusable a piece of software is based on a set of attributes. This reusability measurement determines how likely it is that a piece of software may be used in different contexts not previously anticipated with no or few modifications. These metrics are very important on the reuse design and reuse reengineering fields since they guide how parts have to be designed or changed, in case of already existing parts, to become more reusable.

There are different sets of attributes used for reusability prediction referenced in the literature: reuse level of previous lifecycle objects; number of lines of code and comment to code ratio (Frakes and Terry 1996); code complexity, reliability and quality (Poulin 1994).

6.3 Reuse Repository Metrics

Reuse repositories play an important role in reuse programs since they act as a convergence point for all reuse efforts, related to reusable parts production or consumption. The efficiency of reuse repositories in aspects such as availability and quality of search results, for example, may be a decisive factor for a better reuse activity and greater positive impacts on the quality and the cost of the produced software.

Before adopting reuse repositories, organizations must ensure they have a broader reuse strategy, including a well established process, a continuous educational program and proper tool support. Issues regarding the size and amount of reuse repositories must also be defined in advance in order to increase the probability of a successful reuse initiative.

6.4 Chapter Summary

This Chapter presented the existing reuse metrics in details showing their evolutions and strong and weak points using an application example.

The next Chapter surveys an important aspect related to software reuse: the software component certification area. The failure cases that can be found in the literature are also described in this Chapter.

6.5 References

(Barnes et al., 1988) Barnes, B.; Durek, T.; Gaffney, J.; Pyster, A. **A Framework and Economic Foundation for Software Reuse**. In: *IEEE Tutorial: Software Reuse - Emerging Technology*, ed. W. Tracz. Washington, D.C.: IEEE Computer Society Press, 1988.

(Barnes & Bollinger, 1991) Barnes B. & Bollinger T., **Making software reuse cost effective**. In: *IEEE Software*, Vol. 08, No. 01, January, 1991, pp 13-24.

(Basili et al., 1996) Basili, V. R.; Briand, L. C.; Melo, W. L. **How Reuse Influences Productivity in Object-Oriented Systems**. In: *Communications of the ACM*, Vol. 39, No. 10, October, 1996, pp. 104-116

(Curry et al., 1999) Curry, W.; Succi, G.; Smith, M. R.; Liu, E.; Wong, R. **Empirical Analysis of the Correlation between Amount-of-Reuse Metrics in the C Programming Language**. In: *Proceedings of the Fifth Symposium on Software Reusability (SSR)*, pp. 135-140, 1999, ACM Press.

(Devanbu et al., 1996) Devanbu, P. T.; Karstu, S.; Melo, W. L.; Thomas, W. **Analytical and Empirical Evaluation of Software Reuse Metrics**. In: *Proceedings of the 18th International Conference on Software Engineering (ICSE),* Berlin, Germany, pp. 189-199, 1996.

(Frakes & Succi, 2001) Frakes, W. & Succi, G. **An Industrial Study of Reuse, Quality, and Productivity**. In: *Journal of Systems and Software*, Vol. 57, No. 02, June, 2001, pp. 99-106.

(Frakes & Terry, 1994) Frakes, W. & Terry, C. **Reuse Level Metrics**. In: *Proceedings of the 3rd IEEE International Conference on Software Reuse (ICSR): Advances in Software Reusability*, Rio de Janeiro, Brazil, 1994.

(Frakes & Terry, 1996) Frakes, W.; Terry, C. **Software Reuse: Metrics and Models**. In: *ACM Computing Surveys*, Vol. 28, No. 02, June, 1996, pp. 415-435.

(Gaffney and Durek 1989) Gaffney J. E. and Durek T. A., In: **Software reuse - key to enhanced productivity: some quantitative models**. In: *Information and Software Technology*, pp. 258-267, 1989.

(Henry and Faller, 1995) Henry E. and Faller B., **Large-Scale Industrial Reuse to Reduce Cost and Cycle Time**. In: *IEEE Software*, Vol. 12, No. 05, September, 1995, pp. 47-53.

(Lim, 1994) Lim W., **Effects of Reuse on Quality, Productivity, and Economics**. In: *IEEE Software*, Vol. 11, No. 05, September, 1994, pp. 23-30.

(Mascena et al., 2005) Mascena J. C. C. P., Almeida E. S., Meira S. R. L., **A Comparative Study on Software Reuse Metrics and Economic Models from a Traceability Perspective**. In: *The IEEE International Conference on Information Reuse and Integration (IRI)*, Las Vegas, USA, 2005.

(Poulin, 1994) Poulin J. **Measuring Software Reusability**. In: *Proceedings of the 3rd IEEE International Conference on Software Reuse (ICSR): Advances in Software Reusability*, Rio de Janeiro, Brazil, 1994.

(Poulin, 1995) Poulin Jeffrey S., **Populating Software Repositories: Incentives and Domain-Specific Software**. In: *Journal of Systems and Software*, Vol. 30, No. 03, September, 1995, pp. 187-195.

(Poulin, 1996) Poulin Jeffrey S., **Measuring Software Reuse: Principles, Practices and Economic Models**. *Addison Wesley Professional,* November, 1996, pp. 195.

(Poulin, 2002) Poulin, J. **An Agenda for Software Reuse Economics**. In: *Proceedings of the International Workshop on Reuse Economics (IWRE),* Texas, USA, 2002.

(Poulin & Caruso, 1993) Poulin, J.; Caruso, J. **A Reuse Metrics and Return on Investment Model**. In: *Proceedings of the 2nd IEEE Workshop on Software Reuse (WSR): Advances in Software Reusability,* Lucca, Italy, 1993, pp. 152-156.

Chapter 7

Software Component Certification: A Survey

This Chapter presents a survey of the state-of-the-art of software component certification research (Alvaro et al., 2005a), in an attempt to analyze this trend in CBSE/CBD and to probe some of the component certification research directions. In this way, works related to the certification process in order to evaluate software component quality are surveyed. However the literature contains several works related to software component quality achievement, such as: component testing (Councill, 1999), (Beydeda & Gruhn, 2003), component verification (Wallin, 2002), component contracts (Beugnard et al., 1999), (Reussner, 2003), among others (Kallio & Niemela, 2001), (Cho et al., 2001). Since the focus of this survey is on processes for assuring component quality, it does not cover these works, which deal only with isolated aspects of component quality.

Existing literature is not that rich in reports related to practical software component certification experience, but some relevant research works explore the theory of component certification in academic scenarios. In this sense, this Chapter presents a survey of software component certification research, from the early 90's up to today. The timeline can be "divided" into two ages: from 1993 to 2001 the focus was mainly on mathematical and test-based models and after 2001 the researches focused on techniques and models based on predicting quality requirements.

7.1 Early Age: Mathematical and Test-Based Models

Most research published in this period focus on mathematical and test-based models. In 1993, Poore et al. (Poore et al., 1993) developed an approach based on the usage of three mathematical models (sampling, component and certification models), using test cases to report the failures of a system later analyzed in order to achieve a reliability index. Poore et al. were concerned with estimating the reliability of a complete system, and not just the reliability of individual software units, although, they did consider how each component affected the system reliability.

After that, in 1994, Wohlin & Runeson (Wohlin & Runeson, 1994) presented the first method of component certification using modeling techniques, making it possible not only to certify components but to certify the system containing the components as well. The method is composed of the usage model and the usage profile. The usage model is a structural model of the external view of the components, complemented with a usage profile, which describes the actual probabilities of different events that are added to the model. The failure statistics from the usage test form the input of a certification model, which makes it possible to certify a specific reliability level with a given degree of confidence.

An interesting point of this approach is that the usage and profile models can be reused in subsequent certifications, with some adjustments that may be needed according to each new situation. However, even reusing those models, the considerable amount of effort and time that is needed makes the certification process a hard task.

Two years later, in 1996, Rohde et al. (Rohde et al., 1996) provided a synopsis of in-progress research and development in reuse and certification of software components at Rome Laboratory of the US Air Force, where a Certification Framework (CF) for software components was being developed. The purpose of the CF was: to define the elements of the reuse context that are important to certification; to define the underlying models and methods of certification; and, finally, to define a decision-support technique to construct a context-sensitive process for selecting and applying the techniques and tools to certify components. Additionally, Rohde et al. had developed a Cost/Benefit plan that describes a systematic approach for evaluating the costs and benefits of applying certification technology within a reuse program. After analyzing this certification process, Rohde et al. found some points that should be better formulated in order to increase the certification quality, such as techniques to find errors (i.e. the major errors are more likely to be semantic, not locally visible, rather than syntactic, which this process was looking for) and thus the automatic tools that implement such techniques.

In summary, Rohde et al. considered only the test techniques to obtain the defects result in order to certificate software components. This is only one of the important techniques that should be applied to the component certification.

In 1998, the Trusted Components Initiative (TCI)[1], a loose affiliation of researchers with a shared heritage in formal interface specification, stood out from the pack. Representative of TCI is the use of pre/post conditions on APIs (Meyer, 1997), supporting compositional reasoning, but only about a restricted set of behavioral properties of assemblies. Quality attributes, such as security, performance, availability, and so forth, are beyond the reach of these assertion languages.

The major advanced achievement of TCI was the practical nature of the experiments conducted.

In this same year, Voas (Voas, 1998) defined a certification methodology using automated technologies, such as black-box testing and fault injection to determine if a component fits into a specific scenario.

This methodology uses three quality assessment techniques to determine the suitability of a candidate COTS component. **(i) Black-box component testing** is used to determine whether the component quality is high enough; **(ii) System-level fault injection** is used to determine how well a system will tolerate a faulty component; **(iii) Operational system testing** is used to determine how well the system will tolerate a properly functioning component, since even these components can create system wide problems.

The methodology can help developers to decide whether a component is right for their system or not, showing how much of someone else's mistakes the components can tolerate.

According to Voas, this approach is not foolproof and perhaps not well-suited to all situations. For example, the methodology does not certify that a component can be used in all systems. In other words, Voas focused his approach on certifying a certain component within a specific system and environment, performing several types of tests according to the three techniques that were cited above.

[1] http://www.trusted-components.org

Another work involving component test may be seen in (Wohlin & Regnell, 1998), where Wohlin & Regnell extended their previous research (Wohlin & Runeson, 1994), now, focusing on techniques for certifying both components and systems. Thus, the certification process includes two major activities: **(i)** usage specification (consisting of a usage model and profiles) and **(ii)** certification procedure, using a reliability model.

The main contribution of that work is the division of components into classes for certification and the identification of three different ways for certifying software systems: **(i) Certification process**, in which the functional requirements implemented in the component are validated during usage-based testing in the same way as in any other testing technique; **(ii) Reliability certification of component and systems**, in which the component models that were built are revised and integrated to certificate the system that they form; and, **(iii) Certify or derive system reliability**, where the focus is on reusing the models that were built to certify new components or systems.

In this way, Wohlin & Regnell provided some methods and guidelines for suitable directions to support software component certification. However, the proposed methods are theoretical without experimental study. According to Wohlin & Regnell, "*both experiments in a laboratory environment and industrial case studies are needed to facilitate the understanding of component reliability, its relationship to system reliability and to validate the methods that were used only in laboratory case studies*" (pp. 09). Until now, no progress in these directions has been achieved.

The state of the art, up to around 1999, was that components were being evaluated only with the results of the tests performed in the components. However, there was no well-defined way to measure the efficiency of the results. In 2000, Voas & Payne (Voas & Payne, 2000) defined some dependability metrics in order to measure the reliability of the components, and proposed a methodology for systematically increasing dependability scores by performing additional test activities. This methodology helps to provide better quality offerings, by forcing the tests to only improve their score if the test cases have a greater tendency to reveal software faults. Thus, these metrics and methodology do not consider only the number of tests that a component received but also the "fault revealing" ability of those test cases. This model estimates the number of test cases necessary in order to reveal the seeded errors. Beyond this interesting point, the Voas & Payne work was applied to a small amount of components in an academic scenario. Even so, the methodology presented some limitations, such as: the result of the “fault revealing” ability was not satisfactory; the metrics needed more precision; and, there was a lack of tools to automate the process. Additionally, this methodology was not applied to the industry, which makes its evaluation difficult.

In 2001, Morris et al. (Morris et al., 2001) proposed an entirely different model for software component certification. The model was based on the tests that developers supply in a standard portable form. So, the purchasers can determine the quality and suitability of purchased software.

This model is divided into four steps: **(i) Test Specification**, which uses XML (eXtensible Markup Language) files to define some structured elements that represent the test specifications; **(ii) Specification Document Format**, which describes how the document can be used or specified by a tester; **(iii) Specified Results**, which are directly derived from a component’s specification. These results can contain an exact value or a method for computing the value, and are stored in the test specifications of the XML elements; and, **(iv) Verificator**, which evaluates a component. In other words, Morris built a tool that reads these XML files and performs the respective tests on the components, according to the parameters defined in XML files.

This model has some limitations for software component certification, such as: additional cost for generating the tests, developer resources to build these tests, and the fact that it was conceived only for syntactic errors. However, as cited above, the majority of errors are likely to be semantic, not locally visible, rather than syntactic, which was the aim of the model.

Although this period was mainly focused on mathematical and test-based models, there were different ideas around as well. One work that can be cited was published in 1994. Merrit (Merrit, 1994) presented an interesting suggestion: the use of component certification levels. These levels depend on the nature, frequency, reuse and importance of the component in a particular context, as it follows:

- **Level 1:** A component is described with keywords and a summary and is stored for automatic search. No tests are performed; the degree of completeness is unknown;

- **Level 2:** A source code component must be compiled and metrics are determined;
- **Level 3:** Testing, test data, and test results are added; and
- **Level 4:** A reuse manual is added.

Although simple, these levels represent an initial component maturity model. To reach the next level, the component efficiency and documentation should be improved. The closer it is to level four, the higher the probability that the component is trustable and can be easily reused. Moreover, Merrit begins to consider other important characteristics related to component certification, such as attaching some additional information to components in order to facilitate their recovery, defining metrics to assure the quality of the components, and providing a component reutilization manual in order to help its reuse in other contexts. However, this is just a suggestion of certification levels and no practical work was actually done to evaluate it.

Another work that goes beyond mathematical and test-based models, discussing important issues of certification, was a panel presented in ICSE’2000 - International Conference on Software Engineering, by Heineman et al. (Heineman et al., 2000). The panel had the objective of discussing the necessity of trust assurance in components. CBSE researchers participated in this discussion, and all of them agreed that the certification is essential to increase software component adoption and thus its market. Through certification, consumers may know the trust level of components before acquiring them.

Besides these contributions, the main advance achieved in this period was the fact that component certification began to attract attention and started to be discussed in the main CBSE workshops (Crnkovic et al., 2001), (Crnkovic et al., 2002).

7.2 Second Age: Testing is not enough to assure component quality

After a long time considering only tests to assure component reliability levels, around 2000, the research within the area started to change focus, and other issues began to be considered in component certification, such as reuse level degree, reliability degree, among other properties.

In 2001, Stafford & Wallnau (Stafford & Wallnau, 2001) developed a model for component marketplaces that support prediction of system properties prior to component selection. The model is concerned with the question of verifying functional and quality-related values associated with a component. This work introduced notable changes in this area, since it presents a CBD process with support for component certification according to the credentials, provided by the component developer. Such credentials are associated with arbitrary properties and property values with components, using a specific notation such as <*property*,*value*,*credibility*>. Through credentials, the developer chooses the best components to use in the application development based on the "credibility" level.

Stafford & Wallnau also introduced the notion of active component dossier, in which the component developer packs a component along with everything needed for the component to be used in an assembly. A dossier is an abstract component that defines certain credentials, and provides benchmarking mechanisms that, given a component, will fill in the values of these credentials.

Stafford & Wallnau finalized their work with some open questions, such as: how to certify measurement techniques? What level of trust is required under different circumstances? Are there other mechanisms that might be used to support trust? If so, are there different levels of trust associated with them and can knowledge of these differences be used to direct the usage of different mechanisms under different conditions?

Besides these questions, there are others that must be answered before a component certification process is achieved, some of these are apparently as simple as: what does it mean to trust a component? (Hissam et al., 2003), or as complex as: what characteristics of a component make it certifiable, and what kinds of component properties can be certified? (Wallnau, 2003).

Concurrently, in 2001, Councill (Councill, 2001) examined other aspects of component certification, describing, primarily, the human, social, industrial, and business issues required to assure trusted components. These issues were mainly concerned with questions related to software faults and in which cases these can be prejudicial to people; the cost-benefit of software component certification; the certification advantage to minimize project failures, and the certification costs related to the quantity of money that the companies will save by using this technique. The aspects considered in this work led Councill to assure, along with Heineman (Heineman et al., 2000), (Heineman & Councill, 2001), Crnkovic (Crnkovic, 2001) and Wallnau (Wallnau, 2003), that certification is strictly essential for software components.

In this same year, Woodman et al. (Woodman et al., 2001) analyzed some processes involved in various approaches to CBD and examined eleven potential CBD quality attributes. According to Woodman et al., only six requirements are applicable to component certification: *Accuracy*, *Clarity*, *Replaceability*, *Interoperability*, *Performance* and *Reliability*. But these are "macro-requirements" that must be split into some "micro-requirements" in order to aid the measurement task. Such a basic requirement definition is among the first efforts to specify a set of properties that should be considered when dealing with component certification. However, all of these requirements should be considered and classified in an effective component quality model in order to achieve a well-defined certification process.

In 2002, Comella-Dorda et al. (Comella-Dorda et al., 2002) proposed a COTS software product evaluation process. The process contain four activities, as follows: **(i) Planning the evaluation**, where the evaluation team is defined, the stakeholders are identified, the required resources are estimated and the basic characteristics of the evaluation activity are determined; **(ii) Establishing the criteria**, where the evaluation requirements are identified and the evaluation criteria is constructed; **(iii) Collecting the data**, where the component data is collected, the evaluations plan is done and the evaluation is executed; and **(iv) Analyzing the data**, where the results of the evaluation are analyzed and some recommendations are given.

However, the proposed process is an ongoing work and, until now, no real case study has been carried out in order to evaluate this process, leaving unknown what is really efficient in evaluating software components.

With the same objective, in 2003 Beus-Dukic & Boegh (Beus-Dukic & Boegh, 2003) proposed a method to measure quality characteristics of COTS components, based on the latest international standards for software product quality (ISO/IEC 9126 (ISO/IEC 9126, 2001), ISO/IEC 12119 (ISO/IEC 12119, 1994) and ISO/IEC 14598 (ISO/IEC 14598, 1998)). The method is composed of four steps, as follows: **(i) Establish evaluation requirements**, which include specifying the purpose and scope of the evaluation, and specifying evaluation requirements; **(ii) Specify the evaluation**, which include selecting the metrics and the evaluation methods; **(iii) Design the evaluation**, which considers the component documentation, development tools, evaluation costs and expertise required in order to make the evaluation plan; and **(iv) Execute the evaluation**, which include the execution of the evaluation methods and the analysis of the results.

Although similar to the previous work Comella-Dorda et al. and Beus-Dukic & Boegh works are based on international standards for software product quality, basically the ISO 14598 principles. However, the method proposed was not evaluated in a real case study, and, thus its real efficiency in evaluating software components is still unknown.

In 2003, Hissam et al. (Hissam et al., 2003) introduced Prediction-Enabled Component Technology (PECT) as a means of packaging predictable assembly as a deployable product. PECT is meant to be the integration of a given component technology with one or more analysis technologies that support the prediction of assembly properties and the identification of the required component properties and their possible certifiable descriptions. This work, which is an evolution of Stafford & Wallnau's work (Stafford & Wallnau, 2001), attempts to validate the PECT and its components, giving credibility to the model, which will be further discussed in this section.

Another approach was proposed by McGregor et al. in 2003 (McGregor et al., 2003), defining a technique to provide component-level information to support prediction of assembly reliabilities based on properties of the components that form the assembly. The contribution of this research is a method for measuring and communicating the reliability of a component in a way that it becomes useful to describe components intended to be used by other parties. The method provides a technique for decomposing the specification of the component into logical pieces about which it is possible to reason.

In McGregor et al.'s work, some "roles" (component services) are identified through the component documentation and what the developer may have listed as roles, identifying the services that participate in those roles. The reliability test plan identifies each of the roles and, for each role, the services that implement the role, providing reliability information about each role that the component is intended to support. However, this method is not mature enough to have its real efficiency and efficacy evaluated in a proper way. According to McGregor et al., this method is a fundamental element in an effort to construct a PECT (Hissam & Wallnau, 2003).

During 2003, a CMU/SEI's report (Wallnau, 2003) extended Hissam & Wallnau's work (Hissam & Wallnau, 2003), describing how component technology can be extended in order to achieve Predictable Assembly from Certifiable Components (PACC). This new initiative is developing technology and methods that will enable software engineers to predict the runtime behavior of assemblies of software components from the properties of those components. This requires that the properties of the components are rigorously defined, trusted and amenable to certification by independent third parties.

SEI's approach to PACC is PECT, which follows Hissam & Wallnau work (Hissam & Wallnau, 2003). PECT is an ongoing research project that focuses on analysis – in principle any analysis could be incorporated. It is an abstract model of a component technology, consisting of a construction framework and a reasoning framework. In order to concretize a PECT, it is necessary to choose an underlying component technology, to define restrictions on that technology to allow predictions, and to find and implement proper analysis theories. The PECT concept is portable, since it does not include parts that are bound to any specific platform. Figure 7.1 shows an overview of this model.

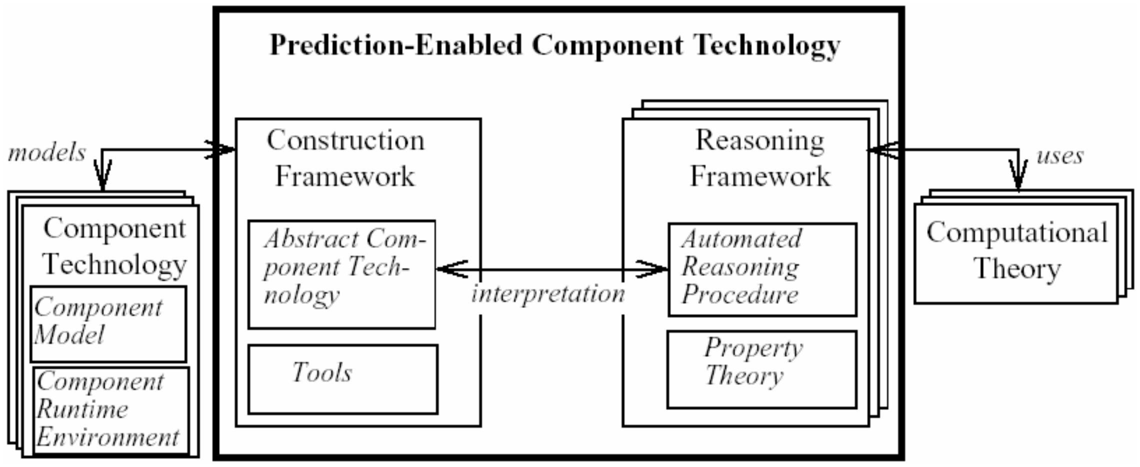


Figure 7.1. Structure of Prediction-Enabled Component Technology (Wallnau, 2003).

A system built within the PECT framework can be difficult to understand, due to the difficulty of mapping the abstract component model into the concrete component technology. It is even possible that systems that look identical at the PECT level behave differently when realized on different component technologies.

Although PECT is highly analyzable and portable, it is not very understandable. In order to understand the model, the mapping to the underlying component technology must be understood as well.

This is an ongoing work in the current SEI research framework. This model requires a better maturation by the software engineering community in order to achieve trust on it. Therefore, some future works are being accomplished, such as: tools development to automate the process, the applicability analysis of one or more property theories, non-functional requirements certification, among other remarks. Moreover, there is still the need for applying this model in industry scenarios and evaluating the validity of the certification.

In another work, in 2003, Meyer (Meyer, 2003) highlighted the main concepts behind a trusted component along two complementary directions: a "low road", leading to certification of existing components (e.g. defining a component quality model), and a "high road", aimed at the production of components with fully proved correctness properties. In the first direction, Meyer was concerned with establishing the main requirements that a component must have. Meyer's intention is to define a component quality model, in order to provide a certification service for existing components – COM, EJB, .NET, OO libraries. This model - still under development - has five categories. When all properties in one category are achieved, the component has the corresponding quality level.

In the second direction, Meyer analyzed the previous work in order to construct a model that complements its certification process. The intention is to develop components with mathematically proved properties.

However, these two directions are still ongoing research. The effort to develop a component certification standard is only in its beginning.

7.3 Failures in Software Component Certification

The previous section presented a survey related to component certification research. This section describes two failure cases that can be found in the literature. The **first** failure occurred in the US government, when trying to establish criteria for certificating components, and the **second** failure happened with an IEEE committee, during an attempt to obtain a component certification standard.

(i) Failure in National Information Assurance Partnership (NIAP). One of the first initiatives attempting to achieve trusted components was the NIAP. The NIAP is an U.S. Government initiative originated to meet the security testing needs of both information technology (IT) consumers and producers. NIAP is collaboration between the National Institute of Standards and Technology (NIST) and the National Security Agency (NSA). It combines the extensive IT security experience of both agencies to promote the development of technically sound security requirements for IT products, systems and measurement techniques.

Thus, from 1993 until 1996, NSA and the NIST used the Trusted Computer Security Evaluation Criteria (TCSEC), a.k.a. "Orange Book."[2], as the basis for the Common Criteria[3], aimed at certifying security features of components. Their effort was not crowned with success, at least partially because it had defined no means of composing criteria (features) across classes of components and the support for compositional reasoning, but only for a restricted set of behavioral assembly properties (Hissam & Wallnau, 2003).

(ii) Failure in IEEE. In 1997, a committee was gathered to work on the development of a proposal for an IEEE standard on software components quality. The initiative was eventually suspended, in this same year, since the committee came to a consensus that they were still far from getting to the point where the document would be a strong candidate for a standard (Goulao & Abreu, 2002).

7.4 Summary of the Study

Figure 7.2 summarizes the timeline of research on the software component certification area, where the dotted line marks the main change in this research area, from 1993 to

[2] http://www.radium.ncsc.mil/tpep/library/tcsec/index.html
[3] http://csrc.nist.gov/cc

2004 (Figure 7.2.) Besides, there were two projects that failed (represented by an “X”), one project that was too innovative for its time (represented by a circle) and two projects related to certification concepts, its requirements and discussion about how to achieve component certification (represented by a square). The arrows indicate that a work was extended by another.

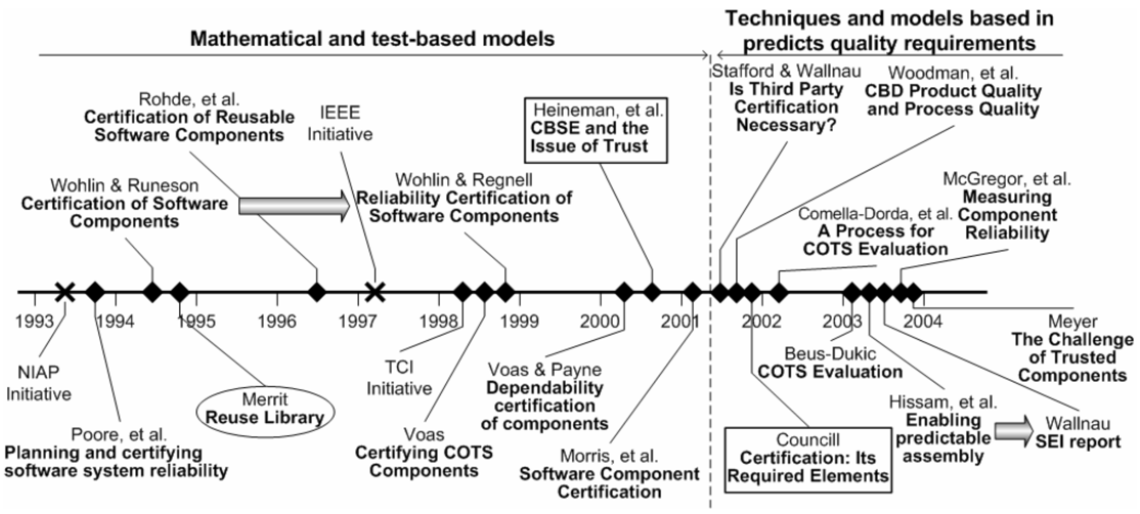

Mathematical and test-based models
Techniques and models based in predicts quality requirements
Rohde, et al. Certification of Reusable Software Components
IEEE Initiative
Heineman, et al. CBSE and the Issue of Trust
Stafford & Wallnau Is Third Party Certification Necessary?
Woodman, et al. CBD Product Quality and Process Quality
Wohlin & Runeson Certification of Software Components
Wohlin & Regnell Reliability Certification of Software Components
McGregor, et al. Measuring Component Reliability
Comella-Dorda, et al. A Process for COTS Evaluation
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
NIAP Initiative
Poore, et al. Planning and certifying software system reliability
Merrit Reuse Library
TCI Initiative
Voas Certifying COTS Components
Voas & Payne Dependability certification of components
Morris, et al. Software Component Certification
Beus-Dukic COTS Evaluation
Councill Certification: Its Required Elements
Hissam, et al. Enabling predictable assembly
Wallnau SEI report
Meyer The Challenge of Trusted Components

Figure 7.2. Research on software component certification timeline.

The research in the component certification area follows two main directions based on: **(i) Formalism:** How to develop a formal way to predict component properties? (e.g. PECT model) and How to build components with fully proved correctness properties? (e.g. Meyer’s “high road” model); and **(ii) Component Quality Model:** How to establish a well-defined component quality model and what kinds of component properties can be certified? (e.g. Meyer’s “low road” model).

However, these works still need some effort to conclude the proposed models and to prove its trust, and needs a definition on which requirements are essential to measure quality in components. Even so, a unified and prioritized set of CBSE requirements for reliable components is a challenge in itself (Schmidt, 2003).

7.5 Chapter Summary

This Chapter presented a survey related to the state-of-the-art of software component certification research. Some approaches found in the literature, including the failure cases, were described. Through this survey, it can be noticed that software component certification is still immature and further research is needed in order to develop processes, methods, techniques, and tools aimed at obtaining well-defined standards for component certification.

The next Chapter presents a discussion on the reuse repository's roles in reuse processes, shows the main types of existent solution and, finally, presents a set of desired requirements for a reuse repository.

7.6 References

(Alvaro et al., 2005) Alvaro, A.; Almeida, E. S.; Meira, S. R. L. **A Software Component Certification: A Survey**, In: *The 31st IEEE EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA), Component-Based Software Engineering (CBSE) Track*, Porto, Portugal, 2005.

(Beugnard et al., 1999) Beugnard, A.; Jezequel, J.; Plouzeau, N.; Watkins, D. **Making component contract aware**, In: *IEEE Computer*, Vol. 32, No. 07, July, 1999, pp. 38-45

(Beus-Dukic & Boegh, 2003) Beus-Dukic, L.; Boegh, J. **COTS Software Quality Evaluation**, In: *The 2nd International Conference on COTS-Based Software System (ICCBSS)*, Lecture Notes in Computer Science (LNCS), Springer-Verlag, Canada, 2003.

(Beydeda & Gruhn, 2003) Beydeda, S.; Gruhn, V. **State of the art in testing components**, In: *The 3th IEEE International Conference on Quality Software (ICQS)*, USA, 2003.

(Cho et al., 2001) Cho, E. S.; Kim, M. S.; Kim, S. D. **Component Metrics to Measure Component Quality**, In: *The 8th IEEE Asia-Pacific Software Engineering Conference (APSEC)*, pp. 419-426, 2001.

(Comella-Dorda et al., 2002) Comella-Dorda, S.; Dean, J.; Morris, E.; Oberndorf, P. **A Process for COTS Software Product Evaluation**, In: *The 1st International Conference on COTS-Based Software System (ICCBSS)*, Lecture Notes in Computer Science (LNCS), Springer-Verlag, USA, 2002.

(Councill, 2001) Councill, B. **Third-Party Certification and Its Required Elements**, In: *The 4th Workshop on Component-Based Software Engineering (CBSE)*, Lecture Notes in Computer Science (LNCS), Springer-Verlag, Canada, May, 2001.

(Councill, 1999) Councill, W. T. **Third-Party Testing and the Quality of Software Components**, In: *IEEE Computer*, Vol. 16, No. 04, July/August, 1999, pp. 55-57.

(Crnkovic et al., 2002) Crnkovic, I.; Schmidt H.; Stafford J.; Wallnau K. C. **Proc. of the 5th Workshop on Component-Based Software Engineering(CBSE): Benchmarks for Predictable Assembly**, In: *The Software Engineering Notes*, Vol. 27, No. 05, May, 2002.

(Crnkovic et al., 2001) Crnkovic, I.; Schmidt H.; Stafford J.; Wallnau K. C. **Proc. of the 4th Workshop on Component-Based Software Engineering(CBSE): Component Certification and System Prediction**, In: *The Software Engineering Notes*, Vol 26, No. 06, May, 2001.

(Crnkovic, 2001) Crnkovic, I. **Component-based software engineering - new challenges in software development**, In: *Software Focus*, Vol. 02, No. 04, December, 2001, pp. 27-33.

(Goulao & Abreu, 2002) Goulao, M.; Abreu, F.B. **The Quest for Software Components Quality**, In: *The 26th IEEE Annual International Computer Software and Applications Conference (COMPSAC)*, England, pp. 313-318, 2002.

(Heineman & Councill, 2001) Heineman, G. T.; Councill, W. T. ***Component-Based Software Engineering: Putting the Pieces Together***, Addison-Wesley, USA, 2001.

(Heineman et al., 2000) Heineman, G. T.; Councill, W.T.; Flynt, J. S.; Mehta, A.; Speed, J. R.; Shaw, M. **Component-Based Software Engineering and the Issue of Trust**, In: *The 22th IEEE International Conference on Software Engineering (ICSE)*, Canada, pp. 661-664, 2000.

(Hissam et al., 2003) Hissam, S. A.; Moreno, G. A.; Stafford, J.; Wallnau, K. C. **Enabling Predictable Assembly**, In: *Journal of Systems and Software*, Vol. 65, No. 03, March, 2003, pp. 185-198.

(ISO/IEC 9126, 2001) ISO 9126, **Information Technology – Product Quality – Part1: Quality Model**, *International Standard ISO/IEC 9126, International Standard Organization (ISO)*, 2001.

(ISO/IEC 12119, 1994) ISO 12119, **Software Packages – Quality Requirements and Testing**, *International Standard ISO/IEC 12119, International Standard Organization (ISO)*, 1998.

(ISO/IEC 14598, 1998) ISO 14598, **Information Technology – Software product evaluation -- Part 1: General Guide**, *International Standard ISO/IEC 14598, International Standard Organization (ISO)*, 1998.

(Kallio & Niemelã, 2001) Kallio, P.; Niemelä, E. **Documented quality of COTS and OCM components**, In: *The 4th Workshop on Component-Based Software Engineering (CBSE)*, Lecture Notes in Computer Science (LNCS) Springer-Verlag, USA, 2001.

(McGregor et al., 2003) McGregor, J. D.; Stafford, J. A.; Cho, I. H. **Measuring Component Reliability**, In: *The 6th Workshop on Component-Based Software Engineering (CBSE)*, Lecture Notes in Computer Science (LNCS) Springer-Verlag, USA, pp. 13-24, 2003.

(Merrit, 1994) Merrit, S. **Reuse Library**, In: *Encyclopedia of Software Engineering*, J.J. Marciniak (editor), John Willey & Sons, pp. 1069-1071, 1994.

(Meyer, 2003) Meyer, B. **The Grand Challenge of Trusted Components**, In: *The 25th IEEE International Conference on Software Engineering (ICSE)*, USA, pp. 660–667, 2003.

(Meyer, 1997) Meyer, B. ***Object-Oriented Software Construction***, 2th Edition Prentice Hall, London, 1997.

(Morris et al., 2001) Morris, J.; Lee, G.; Parker, K.; Bundell, G. A.; Lam, C. P.; **Software Component Certification**. In: *IEEE Computer*, Vol. 34, No. 09, September, 2001, pp. 30-36.

(Poore et al., 1993) Poore, J.; Mills, H.; Mutchler, D. **Planning and Certifying Software System Reliability**, In: *IEEE Computer*, Vol. 10, No. 01, January, 1993, pp. 88-99.

(Reussner, 2003) Reussner, R. H. **Contracts and quality attributes of software components**, In: *The 8th International Workshop on Component-Oriented Programming (WCOP) in conjunction with the 17th ACM European Conference on Object Oriented Programming (ECCOP)*, 2003.

(Rohde et al., 1996) Rohde, S. L.; Dyson, K. A.; Geriner, P. T.; Cerino, D. A. **Certification of Reusable Software Components: Summary of Work in Progress**, In: *The 2nd IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, Canada, pp. 120-123, 1996.

(Schmidt, 2003) Schmidt, H. **Trustworthy components: compositionality and prediction**. In: *Journal of Systems and Software*, Vol. 65, No. 03, March, 2003, pp. 215-225.

(Stafford & Wallnau, 2001) Stafford, J.; Wallnau, K. C. **Is Third Party Certification Necessary?**, In: *The 4th Workshop on Component-Based Software Engineering (CBSE)*, Lecture Notes in Computer Science (LNCS) Springer-Verlag, Canada, 2001.

(Voas, 1998) Voas, J. M. **Certifying Off-the-Shelf Software Components**, In: *IEEE Computer*, Vol. 31, No. 06, 1998, pp. 53-59.

(Voas & Payne, 2000) Voas, J. M.; Payne, J. **Dependability Certification of Software Components**, In: *Journal of Systems and Software*, Vol. 52, No. 02-03, June, 2000, pp. 165-172.

(Wallin, 2002) Wallin, C. ***Verification and Validation of Software Components and Component Based Software Systems***, Building Reliable Component-Based Systems, I. Crnkovic, M. Larsson (editors), Artech House Publishers, July, pp. 29-37, 2002.

(Wallnau, 2003) Wallnau, K. C. **Volume III: A Technology for Predictable Assembly from Certifiable Components**. In: *Software Engineering Institute (SEI), Technical Report*, Vol. III, April, 2003.

(Wohlin & Runeson, 1994) Wohlin, C.; Runeson, P. **Certification of Software Components**, In: *IEEE Transactions on Software Engineering*, Vol. 20, No. 06, July, 1994, pp 494-499.

(Wohlin & Regnell, 1998) Wohlin, C.; Regnell, B. **Reliability Certification of Software Components**, In: *The 5th IEEE International Conference on Software Reuse (ICSR)*, Canada, pp. 56-65, 1998.

(Woodman et al., 2001) Woodman, M.; Benebiktsson, O.; Lefever, B.; Stallinger, F. **Issues of CBD Product Quality and Process Quality**, In: *The 4th Workshop on Component-Based Software Engineering (CBSE)*, Lecture Notes in Computer Science (LNCS), Springer-Verlag, Canada, 2001.

Chapter 8
Reuse Repositories

As heavily stressed in Chapter 2, software reuse has grown in importance and has become an indispensable requirement for companies' competitiveness. Some experiences in industry (Bauer, 1993), (Griss, 1994), (Joos, 1994), (Griss, 1995) have reinforced the idea that software reuse improves productivity and helps to obtain low costs and high quality during the whole software development cycle.

However, an effective application of both technical and non-technical aspects is crucial to the success of software reuse programs. The non-technical aspects include question such as education, training, incentives and organizational management. On the other hand, technical aspects comprise, among other things, the creation of a software reuse environment that supports software engineers and other users in the process of developing software **with** and **for** reuse.

In the context of technical aspects, there are some studies in the literature involving component repositories, although their focus is, for the most part, on component search and retrieval issues (Mili, 1998). Different aspects, such as the roles a repository plays in a reuse process, have not been properly explored so far. As a consequence, some questions raised by companies that desire to adopt a reuse repository remain unanswered. Such questions often include: What are the main requirements of a reuse repository? What kind of artifacts should be stored? What are the practical alternatives?

Under such motivation, this chapter presents a discussion on the roles a reuse repository plays in a reuse process, going through the main types and existing solutions. It finishes up by listing a comprehensive set of desirable requirements for a reuse repository and considerations about the rationale for including each one of the requirements to the list.

8.1 Repository roles

Initially, it is important to understand where a repository appears in the reuse process context, what are its desirable roles and which people should be involved (repository stakeholders).

In any reuse-based development process, it is meaningful to distinguish at least two activities: development ***with*** reuse and development ***for*** reuse. A good example of a reuse-based technology in which these activities are well-defined is the Component-Based Development (CBD) discussed in Chapter 3. For this reason, CBD will be used to illustrate more clearly the reuse repository roles in a reuse-based development process.

According to Apperly (Apperly, 2001), in CBD we have a produce-manage-consume process and the following activities are identified:

- **Production**: focuses on producing and publishing components (development *for* reuse);
- **Management**: focuses basically on organization, availability and quality control; and
- **Consumption**: concentrates on finding and using components (development *with* reuse).

In general, production and consumption activities are executed by developers identified as **component producers** and **component consumers** respectively; and management activities may be executed, for instance, by **project managers**.

Therefore, in this context, there is an essential requirement to build a repository not only for component storage, but also as a tool to support component publication and consumption. In other words, the repository should contain services that satisfy concerns of three stakeholders: producer, consumer and managers. Figure 8.1 shows where the reuse repository appears in the produce-manage-consume process.

8.1.1 Repository as a communication bus

As shown in Figure 8.1, repositories should serve as a communication bus in two ways: 1 - development for reuse (production) and 2 - development with reuse (consumption). This role establishes that a reuse repository must be not only a static base of software components, but also must provide services which allow the interaction between component producers and consumers.

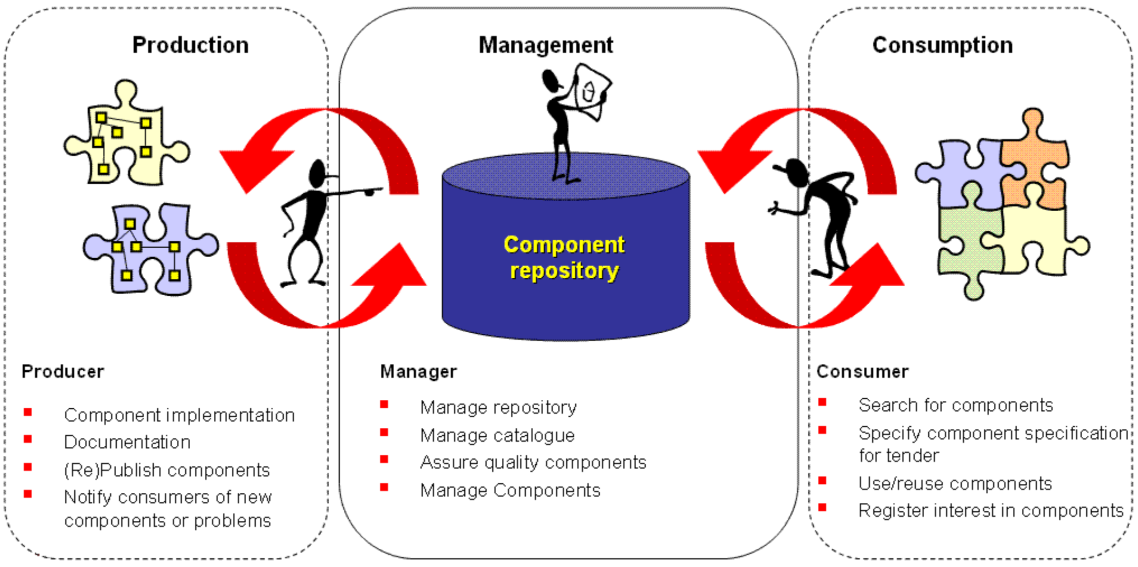


Figure 8.1. Produce-manager-consumer process.

In (Apperly, 2001), an analogy is made between component repositories and warehouse or builders' yard used in construction. Such an analogy reinforces the idea that a repository plays a pivotal role as components are traded between producers, who are interested in publishing or selling their components, and consumers, who are concerned with finding and retrieving components for assembling applications.

8.1.2 Repository as a management assistant

Beyond the important role to serve as a communication bus among producers and consumers, a reuse repository can be used by managers as an assistant when controlling and managing reuse.

Design and code reuse constitute plausible opportunities to boost productivity. However, along with these opportunities comes the need to monitor software reuse at the organizational level (Banker et al., 1993). In fact, software reuse activities span multiple projects and application systems. In order to manage such activities, two fundamental tasks must be employed at the organization level: software monitoring and metrics collection.

Motivated by this problem, Banker et al. proposed the repository-based CASE environments as a solution to turn metrics collection into a tool to be used in the analysis of software reuse at the repository level - what they called "*repository evaluation*". Intending to prove repository evaluation's efficiency, they presented a research that tracked the reuse levels over two years in two large firms using a repository-based CASE tool that maintains the companies' software and all relevant information about that software.

The obtained results indicate that the use of software repositories integrated with CASE environments may significantly help the reuse management at the organizational level and at the same time makes it possible to answer such questions as:

- What kinds of objects are most likely to be reused?
- Under what conditions is reuse most likely to occur?

So, the reuse repository should also offer services that allow the managers to monitor the software reuse across multiple projects, and to do this it is necessary for the repository to provide a set of metrics.

The metrics analysis can indicate a lot of useful things about reuse and can help managers and developers reduce costs. In (Kernebeck, 1997), Kernebeck offers a useful introduction to component repository management and reusability analysis of software components through metrics. Kernebeck shows that reusability analysis allows us to determine the component reuse effort and helps us in deciding between reusing a given component and beginning a new implementation from scratch.

In short, it is possible to monitor the software reuse activities through component repositories with the goal of reducing costs based on systematic software reuse.

8.2 What can be placed in the repository?

In the previous section, we showed the utility of a repository by illustrating its roles in CBD, in which context the elements stored in the repository are software components. However, a company that begins to build a software reuse repository probably has a high number of reusable software parts such as source code, documentation and design models that may not necessarily represent a software component. Because of this, the next question asked is: **what shall become an element of the reuse repository?**

The most reused software product is the source code, which may be considered the most important final product of development activities. Additionally, any other intermediate product of the software lifecycle may be reused (Krueger, 1992).

In fact, it is well known that the reuse level and types of reusable software items may vary considerably depending on the context and not only the source code can be re-used. In this sense, some researchers have dealt with the definition of reusable artifacts.

D’Souza (D’Souza,1999), for example, defines reusable artifacts as any work part that can be utilized in more than one project. D’Souza considers the following artifacts as reusable candidates:

- Compiled code: executable objects;
- Source code: classes and methods;
- Test cases;
- Models and Projects: frameworks, patterns;
- User interface; and
- Plans, strategy and architectural rules.

So, the elements supported by a repository cannot be limited to only one type of artifact. For that reason, we choose to name the repository a “reuse repository” rather than a “component repository”. The first step to determine what can be placed in a repository is to define a more generic element that represents a reusable software unit.

Basically, most researchers prefer to introduce the term “asset” in regards to the definition of a reusable software unit. In a general sense, an asset is something of value which a person or a company owns. This concept may be adapted to the software context, and then the term “asset” may be defined as a reusable software unit that has a significant value for a company and captures business knowledge (Ezran et al., 2002). Ezran et al. consider a software component as a kind of asset that can be executed.

8.2.1 The asset model

According to Ezran et al., the asset model must include two parts: the asset contents (asset body) and the asset meta-data (asset description).

- **Asset contents:** We may define the asset contents as a set of related artifacts (reusable parts). Each artifact means a software lifecycle product such as an analysis model, design model, some source or executable code, and so on; this set may represent the same piece of software at different abstraction levels.
- **Asset meta-data**: The asset meta-data includes all the required information to allow consumers to find and retrieve the appropriate assets. Examples of asset meta-data are: the asset description, classification, qualification and environment information.

A simplified asset model may be represented by the UML diagram showed in Figure 8.2.

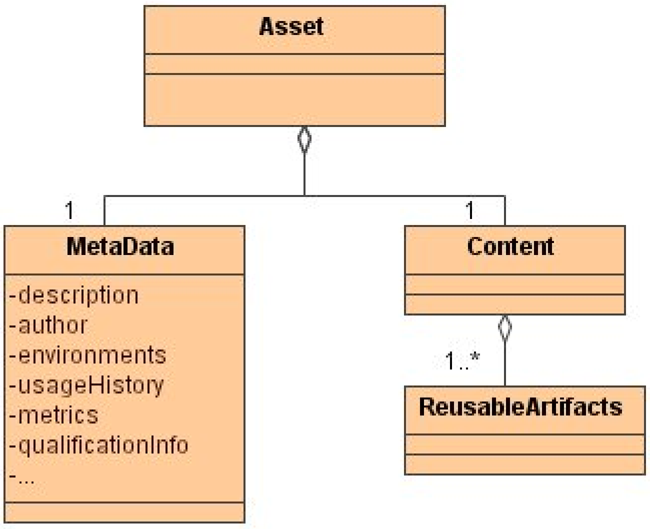


Figure 8.2. Asset model UML Diagram.

It is important to emphasize that the choice of which artifacts will be a part of the asset contents is a subjective process and will vary depending on the value of such artifacts for each company. For example, in (Kernebeck, 1997), Kernebeck emphasizes that it is essential to provide requirement traceability and a test environment within a software component library to cope with the demand of flight-critical software for avionic applications.

Domain analysis and software reengineering are possible options in order to determine the reuse potential of existent software and define the set of assets that will be available in the repository. With this approach, it is necessary to make a domain analysis of the existing software resources within the organization. More detailed domain analysis methods are explained in (Almeida, 2007).

The next Section presents possible tools to be used by companies as “reuse repositories”.

8.3 Practical solutions

In practice, different kinds of tools are used as an option to store reusable assets and to make them available to software developers. Such tools range from the use of configuration management system (CMS) to more elaborate solutions such as component marketplace portals. In fact, practical experiences related in (Ezran el al., 2002) show that most of the analyzed companies prefer to adapt existing traditional software engineering tools to serve as a reuse repository, instead of purchasing reuse-specific tools.

In this section we present a list of categories that group tools which may be used by companies as an alternative to reusable assets repositories. It is important to stress that some of these tools were not designed as reuse-specific solutions and in some cases it is necessary to make adjustments to allow their usage as a reuse repositories.

8.3.1 Configuration Management System (CMS)

As software configuration management is a vital discipline in large scale software development, configuration management systems are commonly used in software development organizations. So, configuration management systems may be viewed as the first alternative to reuse repositories, since they may be used as a store of reusable artifacts. Examples of tools that can be used for this purpose are: CVS (CVS, 2005), ClearCase (ClearCase, 2005), PVCS (PVCS, 2005), Visual Source Safe (VSourceSafe, 2005) and so on.

Some of these tools provide features such as access control, change control and version management. However, they usually have limited APIs that do not allow for the definition, publication and searching of reusable assets in an organized and efficient way.

According to Ezran et al., another problem with this type of tool is that they manage usually fine-grained elements on a project basis, while reusable assets can be coarse-grained, containing the composite elements of multiple projects.

Due to CMS limitations on reusable assets management, some companies implement additional functionality on top of these tools. This principle is followed by most of the source code search tools described in next section.

8.3.2 Source code search tools

As source code is one of the mostly reused software products, many solutions aimed at searching for code in existing repositories have emerged increasingly in both academy and industry.

The general mechanism of these tools is based on the indexing of source code available in existing CMS repositories, benefiting from the fact that such repositories are extensively used in software development projects and providing a code search service. Some of these tools, such as Codase (Codase, 2005) and Koders (Koders, 2005) are available as services on the internet and have an index of code from multiple open source projects.

In general, these tools use free text search and are based on adaptable and flexible mechanisms where a variety of source code repositories such as version control repository (CVS (CVS, 2005), Subversion (Subversion, 2005)) and shared project repositories on the Internet (SourceForge (SourceForge, 2005), Java.Net (Java.net, 2005)) may be plugged in.

The shortcomings of this kind of tool are that reuse is limited to source code and the problems related to search precision given the nonlinguistic nature of source code that may affect the search matching.

Examples of tools on this category: Koders (Koders, 2005), Codase (Codabe, 2005), CodeFinder (Henninger, 1997), CodeBroker (Ye et al, 2000), MARACATU (Garcia et al., 2005), SPARS-J (Yokomori et al., 2003), Agora (Seacord et al., 1998). Some of these tools have additional features as, for instance, the active search support in CodeBroker, but in general they may be seen as source code search engines.

8.3.3 CASE tool Repositories

This alternative consists of building repositories to integrate CASE tools. According to Ezran at al., such repositories should store files that are exchanged among tools. Blaha et al., for example, consider design models, analysis models, mappings and application report data as examples of reusable files that can be stored in these repositories.

CASE tool repositories can also be used to assist other activities within the software development process. In (Banker et al., 1993), Banker et al. present the idea of repository-based CASE environments and show how they can be used to support reuse management at the organizational level. On the other hand, Blaha et al (Blaha et al, 1998) analyze the use of this kind of repository as an infrastructure that facilitates the reverse engineering tasks.

Given such benefits, we can consider this kind of tool as a good option to improve reuse. However, in agreement with Ezran et al., we think this solution requires that the involved CASE tools have open APIs and it is also necessary to define common protocols and formats for data exchange.

8.3.4 Component Managers

Typically, tools in this category are part of enterprise solutions to manage a catalogue of reusable assets and provide services to publish, browse, search and retrieve such assets. Some of them support a more sophisticated features such as version tracking, dependency management and user notification.

These are examples of reuse-specific tools that implement most of the desired requirements listed in Section 8.4. However, they are costly and in some cases have limited compatibility within particular software development environments.

Examples of these reuse-specific tools are: LogIdex (LogIdex, 2005) from LogicLibrary, MS-Visual Component Manager (MS-VCM, 2005) from Microsoft, and Select Component Manager (SCM, 2005) from Select Software.

8.3.5 Component Marketplace

This category represents the component portals on the internet that are intended to the commercialization of software components or development tools. In general, they have services to search, browse and purchase components and tools.

According to Szyperski (2002), the largest example of such a marketplace is ComponentSource (ComponentSource, 2005), but there are others available, such as Software.net (SoftwareNet, 2005).

These component marketplaces change the way in which software is developed (Ezran el al., 2002), so they require a definition of a robust business model that covers questions related to, for instance, the license types of available components (*pay-per-developer*, *pay-per-upgrade* and so on).

8.3.6 Generic repositories

Other types of general purpose repositories that can be used by companies as reuse repositories include:

8.3.6.1 Shared project repositories

This class of tools represents repositories to share software projects and allow distributed teams to collaborate and coordinate development. In general, they have a simple keyword-based search engine to find projects and are well known in open source communities that share their projects on the internet. Examples of shared project repositories: BerliOS (BerliOS, 2005), Java.Net (Java.net, 2005), SourceForge (SourceForge, 2005).

Some of these repositories, SourceForge for instance, have both a public web site for open source development with maximum transparency and an enterprise version designed for operation behind a corporate firewall.

The main problem of these repositories is the limited support to explore software components, source code or reusable algorithms.

8.3.6.2 Wiki systems

This tool category represents systems that allow users to add contents, as on an Internet forum, and also enable other users to edit the contents. They are generally web-based systems that run on one or more web servers and are very useful to exchange information through collaborative effort.

Hence, they can be used to improve the reuse of existent knowledge including patterns and best practices that can be considered reusable assets. However, as shared project repositories, they do not provide adequate support to explore software components, source code or reusable algorithms.

The next section presents a list of requirements that should be considered when building a reuse repository.

8.4 Towards an efficient reuse repository

There is a vast amount of literature involving component repositories. However, most of the existing work focuses on component search and retrieval issues (Mili, 1998) - as will be discussed in the next Chapter - and do not discuss effective ways of defining the features that must be present in an efficient reuse repository.

Under such motivation, this section attempts to handle this issue by presenting a set of requirements that should be considered when building a reuse repository. The following requirements do not represent a final or complete set of functionalities which must be fully present in all reuse-specific repositories, since they depend on the real necessities of each company. However, we think that the identified requirements can serve as a basis for constructing repositories.

The identified requirements are the following:

8.4.1 Search

The repository must provide search mechanisms that allow users to find assets that meet their needs. The search form implemented by these mechanisms can be a combination of the following types:

- **Free-text search**: The user must enter a text string and the mechanism should match such string with any part of the assets meta-data. Optionally, the search can also include the asset content;
- **Keyword**: The user must enter a string of text and the mechanism should match the entered string with all keyword associated to the assets in repository; and
- **Facet-based classification:** Each asset is associated to classifiers (OS type, program language and so on) and users are able to search the assets specifying the classifiers in the search filter provided by the repository.

It should be noted that the search requirement is extremely complex and crucial to repository systems. One of the main researchers involved with repositories, Ali Mili (Mili, 1998), summarizes the works involving artifacts search and retrieval mechanisms in the following way: "*Despite several years of active research, the storage and retrieval of software assets in general and programs in particular remains an open problem. While there is a wide range of solutions to this problem, many of which have* *led to operational systems, no solution offers the right combination of efficiency, accuracy, user-friendliness and generality to afford us a breakthrough in the practice of software reuse*”.

8.4.2 Browsing

Browsing offers a simpler view of the catalogued assets within the repository. So, each asset should be grouped in different categories and it must be possible for users to browse assets by category.

8.4.3 Report generation

The repository should provide services to generate reports that allow, among other things, to get an overview of how the repository is being used. Examples of such reports include: reports that indicate the level of user’s collaboration and components reuse activity; most performed searches; most downloaded assets; more recently added assets, and so on.

8.4.4 User notification

Users should be able to register interest in different events with the aim of receiving notification from the repository when, for instance, new assets are added, new versions of existing assets are created, general news, and so on.

8.4.5 Maintenance services

The repository system should implement administrative services that allow the maintenance of the user-inventory and other inventories utilized by the repository such as asset classifiers and asset artifact types.

8.4.6 Version management

The repository should be able to store different asset versions, so developers are able to retrieve the previous versions of an asset and maintain variants (alternative implementations of the same version of the asset).

8.4.7 Dependency management

Users should be able to search by dependency between assets. These dependencies represent relationship such as “uses” or “composed-of”.

8.4.8 Publish specification

Users should be able to publish only the asset specification (asset meta-data without asset content). The goal of publishing just the specification is to allow developers (reusable assets producers), that register interest in implementing such assets, to be notified when new demands arrive at the repository.

8.4.9 Feedback services

Users should be able to provide feedback about the assets they are using. The feedback services permit the identification of well-evaluated assets and also the tracking of asset usage (the context in which the assets are being used, the problems found when using the assets, and so on). It is also necessary to provide services to assure the quality of available assets.

8.4.10 IDE integration

The most common functionalities of the repository should be available within the developer's programming environment. A repository plug-in to an integrated development environment (Eclipse for instance) is a good option to satisfy this requirement.

8.4.11 Support to multiple repositories

The repository system should support assets from multiple repositories that can be defined in a hierarchical structure.

8.4.12 Access control

The system should have mechanisms to limit user access to system services and repositories, if multiple repositories are supported. So, it should be possible to define different views among users.

8.5 Chapter summary

The process of using existing software artifacts rather than building them from scratch (software reuse) can result in several competitive advantages for companies. However, in order to have the benefits of software reuse, a company has to have mechanisms to store, find and retrieve the reusable software units. In this context, we present the concept of reuse repositories, which represent systems that help software developers and other users to find appropriate reusable software artifacts.

In this chapter, we discussed the roles of a reuse repository in the software development process and introduced the concept of a “software asset”, which represents a reusable software unit to be stored in such repositories. Moreover, a list of practical solutions which can be used as alternatives to reuse-repositories in the form of tools grouped by categories was presented. Finally a set of initial requirements that can be used as basis for building a reuse repository was discussed.

The next chapter presents a brief discussion of component markets, and a survey of the literature on component search. Finally, it presents requirements for efficient component retrieval in such markets.

8.6 References

(Almeida, 2007) Almeida E. **RiSE Approach for Domain Engineering**, Ph.D thesis, Federal University of Pernambuco, Brazil, 2007.

(Apperly, 2001) Apperly H. **Configuration Management and Component Libraries**, In: *Component-Based Software Engineering: Putting the Pieces Together*, Addison Wesley, 2001, pp. 513-526

(Banker et al, 1993) Banker R. D., Kauffman R. J., Zweig D. **Repository Evaluation of Software Reuse**, In: *IEEE Transaction on Software Engineering*, Vol. 19, No 4, April, 1993, pp. 379-389

(Bauer, 1993) Bauer D. **A Reusable Parts Center**, In: IBM Systems Journal, Vol. 32, No. 04, 1993, pp. 620-624.

(BerliOS, 2005) **BerliOS Developer**. Available on https://developer.berlios.de.

(Blaha et al, 1998) Blaha M., LaPlant D., Marvak E. **Requirements for Repository Software**, In: IEEE Software, 1998

(Codase, 2005) **Codase - Source Code Search Engine**. Available on http://www.codase.com, Consulted in October, 2005.

(ComponentSource, 2005) **ComponentSource - The Definitive Source of Software Components**. Available on http://www.componentsource.com, Consulted in October, 2005

(CVS, 2005) **CVS – Open Source Version Control**. Available on http://www.nongnu.org/cvs/, Consulted in October, 2005.

(Ezran el al., 2002) Ezran M.; Morisio, M.; Tully, C. **Practical Software Reuse**, Springer, 2002.

(Frakes & Isoda, 1994) Frakes W.B., Isoda S. **Success Factors of Systematic Software Reuse**, In: *IEEE Software*, Vol. 12, No. 01, September, 1994, pp. 15-19.

(Garcia et al., 2005) Garcia V. C., Durão F., Júnior G. S. A.; Santos M. D. S.; Almeida E. S., Lucrédio D.; Albuquerque J. O.; Meira S. R. L. **Specification, Design and Implementation of a Component Search Engine Architecture** (*in portuguese*), In: *The 5° Workshop de Desenvolvimento Baseado em Componentes (WDBC'2005)*, Brazil, 2005

(Garlan et al, 1995) Garlan D., Allen R., Ockerbloom J. **Architectural Mismatch: Why Reuse is So Hard**, In: *IEEE Software*, Vol. 12, No. 06, November, 1995, pp. 17-26.

(Griss, 1994) Griss, M. L. **Software Reuse Experience at Hewlett-Packard**, In: *Proceedings of the 16th International Conference on Software Engineering*, Sorrento, Italy. IEEE Computer Society,

(Griss, 1995) Griss M. L. **Making Software Reuse Work at Hewlett-Packard**, In: *IEEE Software*, January, 1995.

(Henninger, 1997) Henninger, S. **An Evolutionary Approach to Constructing Effective Software Reuse Repositories**. In: *ACM Transactions on Software Engineering and Methodology*, Vol. 06, No. 02, 1997, pp. 111-140.

(Java.net, 2005) **java.net – The Source for Java Technology Collaboration**. Available on http://www.java.net/, Consulted in October, 2005.

(Joos, 1994) Joos R. **Software Reuse at Motorola**, In: *IEEE Software*, Vol. 11, No. 05, September, 1994, pp. 42-47.

(LogIdex, 2005) **LogicLibrary IT Governance - Service Oriented Architectures SOA : Solutions : Logidex** Available on http://www.logiclibrary.com/solutions/logidex.php, Consulted in October, 2005

(Mili et al, 1998) Mili A., Mili R., Mittermeir R. **A Survey of Software Reuse Libraries**, In: *Annals Software Engineering,* Vol. 05, 1998, pp. 349–414.

(MS-VCM, 2005) **Visual Component Manager Reference** Available on http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vstool1/html/veovrvisualcomponentmanagerreference.asp

(Koders, 2005) **Koders - Source Code Search Engine**. Available on http://www.koders.com/, Consulted in October, 2005.

(Kernebeck, 1997) Kernebeck U. **Component libraries for software re-use**, In: *Microprocessors and Microsystems* 21, pp. 49-54.

(Krueger, 1992) Krueger C.W. **Software Reuse**, In: *ACM Computing Surveys,* Vol. 24, No. 02, June, 1992, pp. 131-183.

(SCM, 2005) **Select Component Manager – active component management and reuse for component based development**. Available on http://www.selectbs.com/products/select-component-manager.htm, Consulted in October, 2005

(Seacord et al., 1998) Seacord R. C., Hissam S. A., Wallnau K. C. **Agora: A Search Engine for Software Components**, In: *IEEE Internet Computing,* Vol. 02, No. 06, November, 1998, pp. 62-70.

(SoftwareNet, 2005) **Store Home - www.software.net**. Available on http://www.software.net, Consulted in October, 2005

(SourceForge, 2005) **VA Software: SourceForge Product Introduction**. Available on http://www.sourceforge.com/, Consulted in October, 2005.

(Szyperski, 1999) Szyperski C. **Component Software: Beyond Object-Oriented Programming**, Addison Wesley, 1999.

(Ye et al, 2000) Ye Y., Fischer G., Reeves B. **Integrating active information delivery and reuse repository systems**, In: *Foundations of Software Engineering,* 2000, pp. 60–68.

(Yokomori et al., 2003) Yokomori R. et al. **Java Program Analysis Projects in Osaka University: Aspect-Based Slicing System ADAS and Ranked-Component**

Search System SPARS-J. In: *International Conference on Software Engineering*, Portland, EUA, pp. 828-829, 2003

Chapter 9
Component Search and Retrieval

Since McIlroy's pioneering work, "*Mass Produced Software Components*" (McIlroy, 1968), the idea of reusing software components in large scale has been pursued by developers and researchers.

Most works follow McIlroy's idea: "the software industry is weakly founded and one aspect of this weakness is the absence of a software component sub-industry". The existence of a market, in which components could be obtained and assembled into applications, was always envisioned.

In the meantime, the literature has many works related to component search and retrieval. However, most of these approaches assume that reuse would take place in-house, with a centralized repository. Even today, with the ideas of component markets coming back to surface, there is still no clear association between search mechanisms and component markets. What role should these mechanisms play in these emerging markets, in order to promote efficient reuse?

This chapter presents a brief discussion on component markets, and a survey of the literature on component search. Finally, it presents requirements for efficient component retrieval in such markets.

9.1 Component Markets

"*Imperfect technology in a working market is sustainable; perfect technology without any market will vanish*" - Clemens Szyperski, in (Szyperski, 1999).

This observation reinforces the fact that components are only viable if the investment in their creation is returned as a result of their deployment. Component-based development methods and tools, distributed component models, and other technologies are useless if there is not a market to consume the produced components.

There are different kinds of component-related markets (Szyperski, 1999), (Bass et al., 2000). Tools to design, assemble and deploy components, component platforms, infrastructure and consulting/integrating services are examples of such markets. And of course, there is a market for components.

Depending on how components are produced and consumed, their return on investment may come in different ways. Considering the following primary stakeholders related to CBSE (Vitharana, 2003):

(i) Component developers: Responsible for producing reusable components. These may be freelance developers, IS departments or even entire organizations specialized in component fabrication;

(ii) Application assemblers: Reuse components and assemble them into applications; and

(iii) Customers: Use the assembled component-based applications.

If the component developers and application assemblers belong to a single organization (for example, one department produces reusable components and other department assembles them into applications), the market is internal, and the investment returns in the form of greater product flexibility, maintainability and reduced time to market, among the other benefits of CBSE.

If the component developers and application assemblers are different organizations (for example, a company specialized in producing and selling components), the market is external. In this case, there is an acquisition process, and the return on investment is direct, in the form of cash.

The internal market is also referred to as in-house reuse. Usually, the assemblers have access to the components code, and may modify it to meet their requirements. This approach is known as white-box reuse. In an external market, the component is usually reused “as is”, i.e. without changes to its code. This kind of reuse is known as black-box reuse.

Each one of these approaches has its benefits and drawbacks. For example, white-box reuse allows full customization, but requires more effort to perform it. In black-box reuse, components are easier to customize, but are also less flexible. In choosing among the reuse strategies, organizations are confronted with a trade-off between component acquisition cost and component customization cost (Ravichandran & Rothenberger, 2003).

In-house white-box reuse is more common. However, as can be seen in (Bass et al., 2000), the component market is growing. In fact, in 1999, Traas & Hillegersberg identified several component vendors on the Internet (Traas & Hillegersberg, 2000). Among these, two stand out: www.componentsource.com and www.flashline.com.

These two component sellers do not make available the components' source code, as well as 85% of the component sellers (Traas & Hillegersberg, 2000). This fact reinforces the idea that black-box reuse is more suitable for such markets. Some authors, such as (Traas & Hillegersberg, 2000), (Ravichandran & Rothenberger, 2003), even state that black-box reuse with component markets could be the silver bullet solution that makes software reuse a reality. However, several issues remain to be resolved, such as support and quality assurance (Ravichandran & Rothenberger, 2003), (Wallnau, 2003) and the difficulty in finding components (Bass et al., 2000), (Ravichandran & Rothenberger, 2003). This last issue is what we discuss in the remainder of this chapter.

9.2 The History of Component Search

A component retrieval mechanism works in the following way, as described[1] in (Mili et al., 1995) (Figure 9.1): When faced by a problem, the reuser understands it in its own way, and then formulates a query, which may be as simple as a set of keywords or as complex as specifications in a formal language. In practice, this first process results in a loss of information, since the reuser is not always capable of exactly understanding the problem, or encoding it in the query language.

To be retrieved, the information about the components must be encoded. This process of classification (also known as indexing), which may be manual or automatic, results in a loss of information. The search itself consists in comparing the query with the indexes and returning the components that match the query. This information loss is the cause of all the effort in this area.

[1] The original description was simplified to facilitate its understanding in the context of this chapter

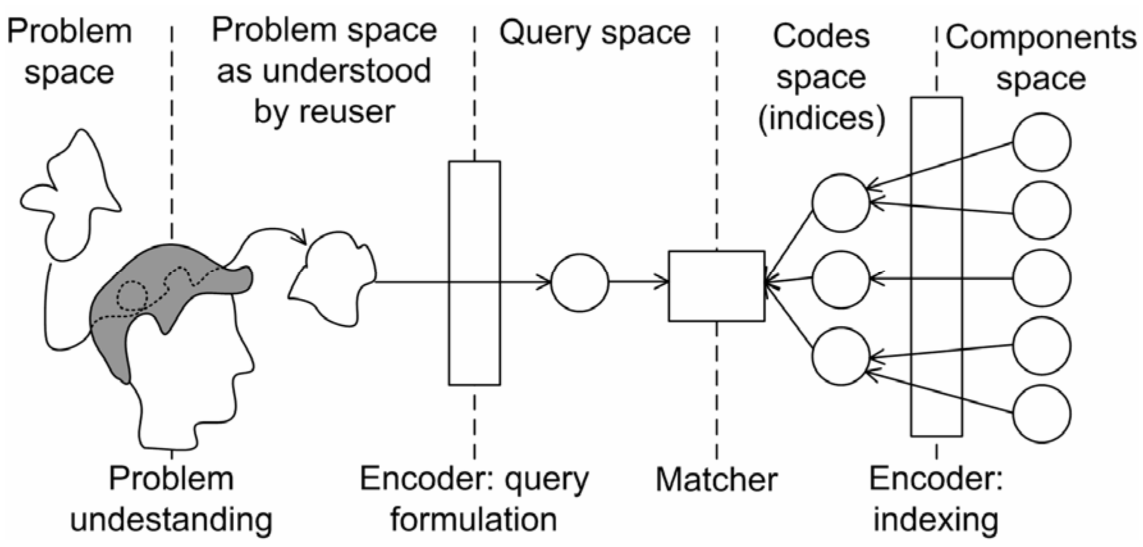


Figure 9.1. Component Retrieval Model.

In the literature, several researches that study efficient ways to retrieve components may be found. These works focus on many aspects of the above model, as can be seen in the next sections.

9.2.1 The classification scheme

Among the works of the early 90's, such as (Prieto-Díaz, 1991), (Maarek *et al.*, 1991), (Podgurski & Pierce, 1993) and (Henninger, 1997), a special focus is placed on the classification schemes used to store software components.

An example of this concern may be seen in (Prieto-Díaz, 1991). In this work, the author proposes the utilization of a facet-based scheme to classify software components (Figure 9.2). In this approach, a limited number of characteristics (facets) that a component may have are defined. Then, a set of possible keywords is associated to each facet. To describe a component, one or more keywords are chosen for each facet.

In the example of Figure 9.2, there are four facets: ***Domain***, ***Functionality***, ***Component Model*** and ***Component Type***. To describe the ***Account*** component, the "***Financial***" keyword was chosen for the first facet, "***Data mining***" and "***Storage***" for the second, and so on.

In this way, it is possible to describe components according to their different characteristics, unlike the traditional hierarchical classifications, where a single node from a tree-based scheme is chosen.

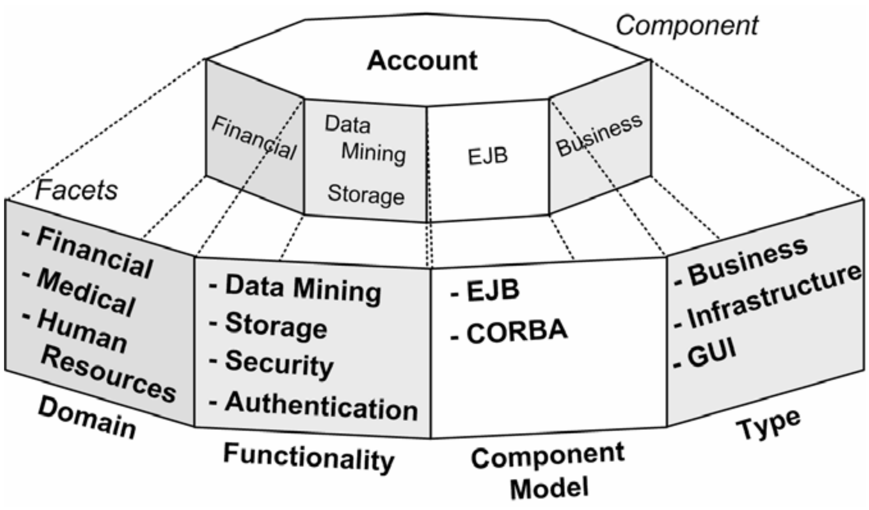


Figure 9.2. Facet-based classification scheme.

The literature of this period also has other approaches to solve the component classification problem. Authors, such as (Maarek et al., 1991), argue that the effort needed to manually classify the components of a software library is too big, since these tend to grow and become huge. Besides, this process is susceptible to the subjective rationale of the person who is responsible for the task of classifying the components. For instance, in a facet-based scheme, two different people may choose different keywords to describe a component.

To overcome this problem, Maarek et al. (Maarek et al., 1991) propose the use of automatic indexing, which consists or automatically extracting, from free-text descriptors, terms or phrases that best describes a component. Statistical and linguistic methods are used to analyze pieces of text (manual pages, for example), identifying and extracting terms to classify the component. Hence, manual effort to describe the components is not needed. Also, divergence of opinion does not exist, since there is no human intervention.

Facet-based classification and statistical analysis are examples of the two main approaches found in the literature of the period:

(i) Controlled vocabulary: The terms used to describe the components are chosen from a fixed set; and

(ii) Uncontrolled vocabulary: Any term can be used to describe the components.

Both approaches have advantages and disadvantages. The uncontrolled vocabulary approach provides more flexibility. When used in automatic indexing, the effort needed to classify the components is minimal, making it very attractive for big software libraries. However, as pointed out by (Prieto-Díaz, 1991), normally software components have little or no free-text descriptors available, a fact which makes such approaches impossible to execute. Besides, when experts are defining a fixed set of terms, such as in facet-based classification, they are already introducing some semantic information, resulting in a more meaningful description.

Although the main research streams in this period go between controlled and uncontrolled vocabularies, other works can also be found, such as (Podgurski & Pierce, 1993). Here, they explore component search through its execution: given a set of input values, and a set of expected output values, the components are executed, and those that produce the expected output are retrieved. This approach results in high precision, but is restricted to simple components, such as routines or pieces of code. Also, performance is affected, since all components must be executed.

9.2.2 Other issues related to component search

In 1994, an empirical study conducted by Frakes & Pole (Frakes & Pole, 1994) showed that the existing classification methods were very similar, when considering precision and recall (commonly used metrics to analyze information retrieval). Then, other issues related to component search started to be considered, such as query formulation and other human aspects.

This concern can be seen in (Henninger, 1994), in which the author discusses the difficulty of elaborating queries, in order to achieve satisfactory results. According to a study performed by the author, the biggest difficulty relies on the distance between what the reuser wants and how components are described. Normally, components are described in terms of their functionality, containing the component's name and “how” the function is performed. However, usually the reuser specifies the query thinking of “what” he wants to do. This difference between “what” and “how” may sometimes be very large, resulting in poor search quality.

Motivated by this idea, the author proposes the combination of two techniques to reduce this problem: *retrieval-by-reformulation* and *spreading activation*. The first attempts to provide means for the reuser to refine his queries, such as storing previously formulated queries to be later consulted and presenting tips to the reuser.

The second technique, *spreading activation*, permits the retrieval of not only components that exactly match the query, but also those that are “closer” to it. This is performed by storing not only information regarding the component, but also the relationships between them.

This idea of a “relaxed” search was already being explored in early works, such as (Prieto-Díaz, 1991) and (Maarek et al., 1991), and it is practically a consensus among researchers, even today.

Another different attempt to improve search quality is presented in (Isakowitz & Kauffman, 1996), where the “relaxed” search is explored together with a hypertext-based navigation mechanism. The process is divided in two steps: first, the reuser elaborates his query, and a set of components that possibly satisfies his need is returned. Second, he browses this set through hypertext-based navigation mechanisms, inspecting the components to find one that suits his needs. By doing this, there is no need to explore the whole repository. Also, another benefit is achieved: after a few search sessions, the reuser will naturally increase his knowledge about the repository, while inspecting the components.

Other interesting approach may be seen in (Henninger, 1997), where the author continues a previous work (Henninger, 1994), proposing a repository that is able to evolve during its usage.

According to the author, different groups of people use different terms to refer to a single concept. In a classification and search mechanism, this is problematic, since the people who classify the components are not the same people who will reuse them (in many cases, they don't even know each other). Therefore, to reduce this gap, a search mechanism should adapt itself to the user.

In this approach, the repository constantly analyzes the queries while they are being formulated. Commonly used terms and successful queries are stored and form a basis to automatically review and rebuild the indexes, so that in future queries, components are more easily found.

9.2.3 The questioning

Despite all these advances, component repositories were still not being widely used. In response to this, the literature of the end of the millennium presents some questioning.

According to (Seacord, 1999), one of the reasons for the historical failure of component repositories comes from their conception as centralized systems. Problems such as excessive control over the stored components, low accessibility and scalability are among the inhibitors of this kind of system. Since then, the idea that repositories should be free and distributed over open networks, to allow large-scale reuse, began to be defended.

In (Seacord et al., 1998), Seacord et al. present the Agora system, which consists in a mechanism that automatically searches and registers components through the Internet, making them available for reuse. The results of this research show that it is possible to use the infrastructure of an open network (the Internet) to promote reuse without the problems associated with centralized repositories.

Another example of the use of repositories based on open networks may be seen in (Silveira, 2000). Here, Silveira introduces the concept of spontaneous software, a web-based object computing paradigm for supporting on-demand, dynamic distribution and integration of distributed reusable software artifacts on user environment during execution time. This model is supported by a general framework, Software Operating System (SOS), which allows software systems to locate, retrieve, install and execute remotely available software artifacts on user desktops.

The framework uses a universal distributed repository, implemented as a set of distributed entities, called containers, which can store the artifacts of any number of authorized producers. The search is performed with the help of a name service (also distributed) for registering, locating and retrieving software artifacts. Clearly, this highly distributed architecture requires extra mechanisms to maintain integrity and consistency. However, there is a gain in performance, due to distributed processing, and reliability, due to the existence of many possible alternative paths.

But not just the nature of the repository (distributed or centralized) was being questioned. In (Hall, 1999), the author shows his doubts about the idea that a big repository is a decisive factor for reuse success. According to him, it is more important to maintain a few components from a specific application domain than a great number of different components. Therefore, an elaborate and complex search mechanism would not be necessary, since there would be only a few components to be inspected. This statement contradicts several researchers, including Victor Basili, an important researcher into the are of software engineering: “Reuse efficiency demands a great catalog of reusable objects” (Caldiera & Basili, 1991).

Delving even deeper into these questions, a report (Bass et al., 2000) from the SEI/CMU (Software Engineering Institute / Carnegie Mellon University) presents a study, conducted between 1999 and 2000, which pursued the reasons that explain why CBSE has not been widely adopted. The study involved several software development organizations and a wide research of the literature, and questioned, among other issues, the real contribution of CASE, including component repositories, to promote the CBSE.

The report concludes that the market perceives the following as key inhibitors of reuse adoption, in decreasing order of importance: a ***lack of available components***, a ***lack of stable standards for component technology***, a ***lack of certified components*** and a ***lack of an engineering method to consistently produce quality systems from components***.

The most important inhibitor is the lack of available components. According to the report, one of the reasons for this to happen may be the difficulty of finding components, which is the subject of this paper.

This SEI report and other studies related to software reuse can be seen in Chapter 2, where several experiences on software reuse are presented.

Independent of these questions, other researchers continue to explore ways to store and retrieve software components from repositories. For example, in (Grundy, 2000), Grundy proposes a new approach to doing this, exporting the properties, events and methods of a component in several units, which represent its different aspects. Each aspect describes what a component provides and what it requires, and is used to connect one component to other (s). The information contained in the aspects is then automatically extracted into a facet-based classification scheme (Prieto-Díaz, 1991), allowing the reuser to formulate queries and to retrieve the components.

Although this approach is based on already explored concepts, such as automatic indexing and facet-based classification, the author introduces a new way to describe the components. According to the author, the use of aspects makes it easier to describe the components, as well as to search for and retrieve them. Besides, by describing what a component provides and requires it is possible to visualize the dependencies between the components, facilitating their maintenance.

A similar idea is used in the CORBA Component Model (CCM) (Omg, 2002). In CCM, it is also possible to describe the provided and required interfaces (facets and receptacles), produced and consumed events. However, in CCM these are divided in four kinds of units, called ports, while in Grundy's approach they are grouped in a single unit: the aspect.

Another approach may be seen in (Zhuge, 2000). This is an example, among others, of the use of formalism to describe a component's behavior. With formalism, it is possible to assure that the retrieved components will fulfill the requirements specified in the query. Besides, the information contained in the component description goes beyond mere isolated keywords, allowing its full semantic to be described without ambiguity. However, it requires both the developer and the reuser to know a formal language, such as POL (Problem-Oriented Language) (Zhuge, 2000).

The use of formalism has always been studied. Other works, such as (Mili et al., 1997) and (Merad & Lemos, 1999), follow the same idea, using formalism to eliminate the ambiguity inherent to an interpretative approach.

9.2.4 The state-of-the-art

One of the most advanced works related to component retrieval may be seen in (Ye & Fischer, 2002). The contributions of this work are important because the Ye & Fischer investigate different issues related to component reuse, involving both its technical and human aspects. According to a study conducted by them, reusers have different levels of knowledge about a software repository, as shown in Figure 9.3 (Ye & Fischer, 2002).

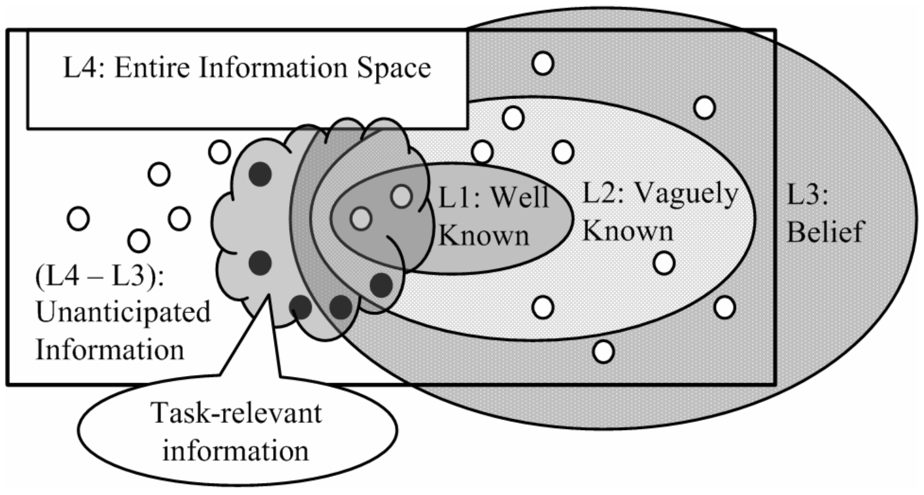

L4: Entire Information Space
L1: Well Known
L2: Vaguely Known
L3: Belief
(L4 – L3): Unanticipated Information
Task-relevant information

Figure 9.3. A reuser's different knowledge levels.

Each oval (L1, L2, L3) represents a set of components that are more or less known by the reuser (Ye & Fischer, 2002). However, only some of the elements are relevant to the problem (the cloud in the Figure). Therefore, ideally, a search mechanism should retrieve only this relevant information.

Their studies also showed that, when faced with a problem, reusers usually start looking for elements in L2 and L3. Elements in L4-L3 are ignored, and the reuse opportunity is lost. Hence, the repository must anticipate the reuser's needs, delivering the information without being requested. This kind of repository is known as an active repository.

More recently, a research work (Garcia et al., 2005) attempts to explore the reuse of open-source software artifacts. Since this kind of artifact is widely available in specialized websites, such as java.net, sourceforge.net and www.tigris.org, this is an excellent opportunity to promote wide-scale reuse.

The idea presented in (Garcia et al., 2005) is to maintain an automatic service, called MARACATU, which continuously scans different open-source repositories in search for reusable artifacts, creating a single index, so that the reuser may perform search on it. When an artifact is found, the reuser may retrieve it from the original source. The approach taken by Garcia et al. is somehow related to the work of Seacord et al. (Seacord et al., 1998), where the focus is to retrieve the artifacts without "permission" from their developers. The mechanism proactively scans the repositories in search for artifacts.

A different approach of reusing open-source artifacts, which takes a more controlled approach, is employed by the website www.koders.com. Here, the open-source artifacts that are available for search and retrieval must be explicitly registered by their developers. A project must be registered with the mechanism in order to make its artifacts available for reuse.

9.3 Towards an efficient mechanism to search and retrieve software components

Figure 9.4 summarizes the history of the research into component search and retrieval methods. According to Figure 9.4, there are some works (represented by an "X") that mark the main changes in this research area. Initially, the focus was placed on the classification scheme, until 1994, when the work of Frakes & Pole (Frakes & Pole, 1994) showed that the different classification methods were similar. Then, other issues started to be considered. In 1998, the idea of an open, distributed repository was first mentioned (Seacord et al., 1998) and (Seacord, 1999).

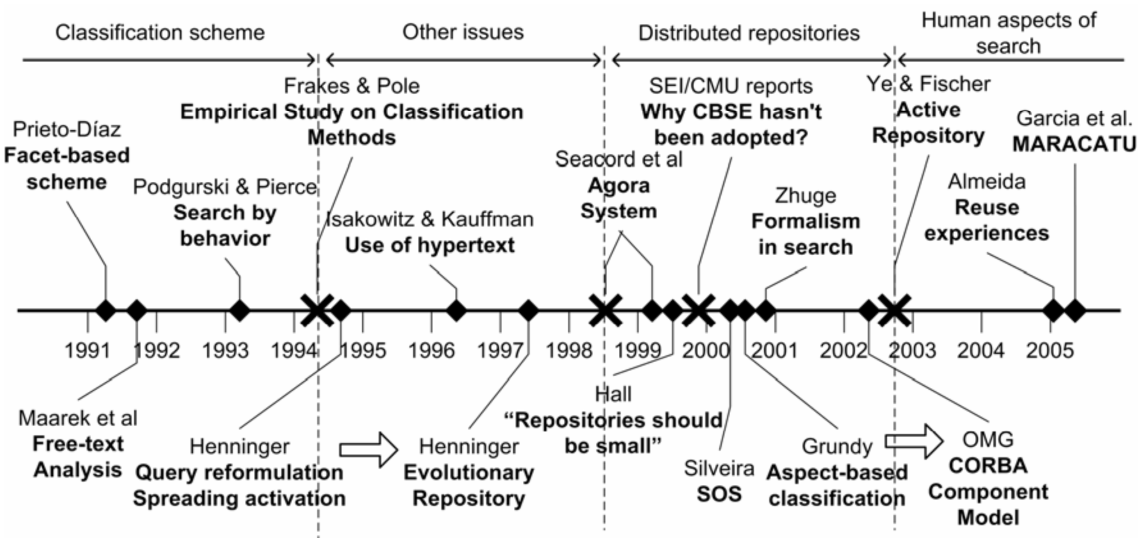

Classification scheme
Other issues
Distributed repositories
Human aspects of search
Prieto-Díaz Facet-based scheme
Podgurski & Pierce Search by behavior
Frakes & Pole Empirical Study on Classification Methods
Isakowitz & Kauffman Use of hypertext
Seacord et al Agora System
SEI/CMU reports Why CBSE hasn't been adopted?
Zhuge Formalism in search
Ye & Fischer Active Repository
Almeida Reuse experiences
Garcia et al. MARACATU
1991 1992 1993 1994 1995 1996 1997 1998 1999 2000 2001 2002 2003 2004 2005
Maarek et al Free-text Analysis
Henninger Query reformulation Spreading activation
Henninger Evolutionary Repository
Hall "Repositories should be small"
Silveira SOS
Grundy Aspect-based classification
OMG CORBA Component Model

Figure 9.4. Research on component search and retrieval.

The SEI/CMU report (Bass et al., 2000) was also important, since it raised several questions regarding CBSE. In 2002, the work of Ye & Fischer (2002) grouped several concepts, both technical and human, into a novel approach for repositories. In 2005, the work of Garcia et al. attempts to promote wide-scale reuse of open-source artifacts, made available on globally distributed repositories.

The time line shown in Figure 9.4 shows the change that has been occurring since the end of the last millennium, leading to the formation of component markets, where components are made available by different vendors, in repositories distributed all over the world. Questions such as the ones raised in (Bass et al., 2000) Ravichandran:cacm:2003 (Ravichandran & Rothenberger, 2003) Szyperski:book:1999 (Szyperski, 1999) also point out the emergence of such markets. Researchers such as (Traas & Hillegersberg, 2000) state that the most appropriate place to buy and sell components would be on the Internet.

Therefore, current works on component search should focus on globally distributed repositories, giving support for component markets, which will deliver reusable software components to developers around the world, as in (Seacord et al.,

1998) and (Silveira, 2000). Then what should be considered when building such mechanisms?

In the next section we try to answer this question by presenting a set of requirements for an efficient mechanism to search for and retrieve software components. We are aware that this is not a definitive set. This is a difficult task, mainly because the idea of component markets is still too immature (Traas & Hillegersberg, 2000) and (Ravichandran & Rothenberger, 2003). However, we believe that the identified requirements constitute a solid basis for future work.

9.3.1 High recall and precision

This is a basic requirement, which must be considered in any search mechanism. High precision means that most relevant elements are retrieved first. High recall means that few relevant elements are left behind.

Several techniques may be used to achieve this. For instance, the approaches that best achieve precision are those employing some kind of formalism, such as (Mili et al., 1997) and (Zhuge, 2000). To achieve a high recall ratio, most prefer to use a “relaxed” search, where not only exact matches are retrieved, but also related elements.

It is important to stress that these measures assume different dimensions when inserted into a global component market, where the number of existing components tends to be huge. Hence, a mechanism with satisfactory precision in a centralized repository may return too much irrelevant information in such scenario.

One interesting observation is made by Szyperski (Szyperski, 1999). Components are always placed within an architectural context: they must refer to some architectural model, domain standard, component framework or platform. However, these all share one property - there are not all that many of these. Thus, a query of these items could drastically reduce the number of components to be examined.

9.3.2 Security

Historically, security has been a minor issue in component search and retrieval. Since in most approaches reuse takes place in-house only, little effort is needed in order to avoid unauthorized access. However, in a globally distributed component market, where people from all over the world have access to these systems, security must be considered a major issue, since there is a wider range of possibilities for an unauthorized person to break security. Certification and authentication techniques, already used in distributed systems, are good candidates to solve this problem.

9.3.3 Query formulation

As already discussed earlier in this Chapter, there is a natural information loss when the reuser is formulating a query. As pointed out by (Henninger, 1994), there is also the conceptual gap between the problem and the solution, since usually components are described in terms of functionality (how), and queries are formulated in terms of the problem (what).

A search mechanism must therefore provide means to help the reuser to formulate the queries, which will consequently reduce this conceptual gap. Examples of such help may be seen in (Henninger, 1994) and (Henninger, 1997).

9.3.4 Component description

During a search, the reuser specifies a query, which is then “matched” against a series of component descriptions. The search mechanism is then responsible for deciding which components are relevant to the reuser. There are two approaches to doing this:

The first one is to use formalism. By formally describing a component's functionality and requiring the reuser to formulate queries in the same formal language, it is possible to automatically check whether or not the component satisfies the reuser's requirements. The other approach is to “roughly” describe components (using facets (Prieto-Díaz, 1991) (Vitharana *et al.*, 2003), or aspects (Grundy, 2000), for example), and to retrieve a set of components that “probably” satisfies the requirements. Then, these components are presented to the reuser, who inspects them and decides which one (s) to retrieve.

Due to some inhibitors, such as the computational cost and the difficulty in stimulating reusers in specifying formal queries, Mili et al. (1995) believe that formal approaches are not practical in this case, being restricted to other application domains such as reactive and real-time systems.

After performing a survey of reuse repositories, Guo & Liki (Guo & Luqi, 2000) noticed that in most cases, the search mechanisms are based on keywords. Some have support for browsing, and only a few use the facets approach. This reflects the fact that, in practice, search engines must be simple and easy to use. However, simple or not, some information must be included in order to help in the reuser's decision:

(i) Interface description: the component's interface must be included, to allow the reuser to identify if the component can or cannot be inserted into his application. It must be stressed that this is not the concept of interface as employed by programming languages, which is usually composed of operation signatures and field descriptors, but a full description of what the component performs and what is required for its execution. This is also known as component contracts (Szyperski, 1999);

(ii) Non-functional requirements: non-functional requirements, such as performance and complexity, must also be included; and

(iii) Vendor / developer: in a competitive market this information may turn out to be very useful.

9.3.5 Repository familiarity

Reuse occurs more frequently with well-known components (Banker et al., 1993), (Ye & Fischer, 2002). Therefore, a search mechanism should help the user to explore and get familiar with the components, so that in future searches it is easier to locate them. In a component market, this is even more important, since it allows reusers to be aware of different components, and different options for a single type of component, facilitating his choices and stimulating the competition among vendors. The literature (in particular, those works presented here) also shows that providing means for the reuser to acquire knowledge about the existing components increases the reuse rate.

There are several ways of achieving this goal. Increasing recall through “relaxed” search is one of them. Henninger's ideas of reformulating queries Henninger (1994) and evolving the repository (Henninger, 1997) also help. Ye & Fischer's active repository (Ye & Fischer, 2002) does this in a novel way, by “teaching” the reuser at the exact moment that the components are needed.

But the technique that seems to be most appropriate to a component market is the hypertext-based browsing (Isakowitz & Kauffman, 1996). Its successful story in the Internet indicates its potential to allow users to rapidly browse through different pages (components), in different web sites (repositories), acquiring knowledge while browsing. Also, the underlying technologies are practically standards, facilitating interoperability, which is the next requirement.

9.3.6 Interoperability

In a scenario involving distributed repositories, it is inevitable to think about interoperability. A search mechanism that operates in such a scenario should be based on standard technologies, in order to facilitate its future expansion and integration with other systems.

Due to the natural heterogeneity of this world-wide distributed scenario, misinterpretation is a constant risk. The simplest way to avoid this is to use standard ways of packaging information. Another way is to package the information together with its original interpretation, as intended by its creator, as happens when using ontologies. According to Szyperski (Szyperski, 1999), the use of ontological approaches is showing potential in the field of component search.

9.3.7 Performance

Performance is usually measured in terms of response time. In centralized systems, the involved variables are the hardware processing power and the search algorithm complexity. In the distributed scenario that is being discussed here, other variables must be considered, such as network traffic, geographical distance and, of course, the great number of available components. Again, Internet-related techniques, such as replication and caching, are examples of how to address such issues.

9.3.8 Summary of the Study

Table 9.1 shows the relationship between the works described in section 9.2 and the requirements from section 9.3. Each number corresponds to a requirement: 1. Recall and Precision, 2. Security, 3. Query Formulation, 4. Component Description, 5. Repository Familiarity, 6. Interoperability, 7. Performance.

Almost every work was concerned with recall and precision ratios and the component description. Other requirements such as query formulation and repository familiarity started to be considered after the middle of the 90's. Security, Interoperability and Performance were only considered by recent works.

Table 9.1. Relationship between the works on component search and the requirements.

Author(s)	1	2	3	4	5	6	7
Prieto-Díaz, 1991	x			x			
Maarek et al., 1991	x			x			
Podgurski & Pierce, 1993	x						
Henninger, 1994	x		x	x	x		
Isakowitz & Kauffman, 1996	x				x		
Henninger, 1997	x		x	x	x		
Seacord et al., 1998	x					x	x
Silveira, 2000		x				x	x
Grundy, 2000	x		x	x			
Zhuge, 2000	x	x	x				
CCM, 2002				x			
Ye & Fischer, 2002	x		x	x	x		
Garcia et al., 2005				x	x	x	x

9.4 Chapter Summary

Today, almost every product can be purchased over the Internet, including books, CDs, and videos, among others. The selling of software components is also included in this list, but is still on its first steps. There are few vendors, and several issues remain unresolved. In this chapter, we cover part of these issues, by discussing the role played by component search mechanisms in component markets.

As demonstrated in some works (Seacord et al., 1998), (Silveira, 2000) and (Garcia et al., 2005), the Internet is a key for enabling such markets. Its underlying technologies are practically standards, and the knowledge acquired during its history must not be discarded.

Many issues discussed here are already addressed by existing web search engines, such as www.google.com. Of course, important issues, such as precision, recall and familiarity with the components are not achieved by those engines. These issues may not seem too serious today, but in larger markets they may become bigger problems.

Reuse is the ability to avoid effort duplication. What a component market offers is effort already spent, in the form of reusable software artifacts. In order to reuse this effort, first it is necessary to find it, and that is what makes an efficient search mechanism essential for promoting large-scale reuse.

Next Chapter, finally, presents some conclusions on the book.

9.5 References

(Almeida et al., 2005) Almeida, E. S. D.;Alvaro, A.; Meira, S. R. D. L. **Key Developments in the Field of Software Reuse**. Submitted to the ACM Computing Surveys, 2005.

(Banker et al., 1993) Banker, R. D.;Kauffman, R. J.;Zweig, D. **Repository Evaluation of Software Reuse**. IEEE Transactions on Software Engineering, Vol. 19, No. 04, 1993, pp. 379-389.

(Bass et al., 2000) Bass, L. *et al.* **Volume I - Market Assessment of Component-Based Software Engineering**. CMU/SEI - Carnegie Mellon University/Software Engineering Institute. 2000.

(Caldiera & Basili, 1991) Caldiera, G.;Basili, V. R. **Identifying and Qualifying Reusable Software Components**. IEEE Computer, Vol. 24, No. 02, 1991, pp. 61-71.

(Frakes & Pole, 1994) Frakes, W. B.;Pole, T. P. **An Empirical Study of Representation Methods for Reusable Software Components**. IEEE Transactions on Software Engineering, Vol. 20, No. 8, 1994.

(Garcia et al., 2005) Garcia, V. C. *et al.* **Specification, Design and Implementation of an Architecture for a Component Search Engine** (*in portuguese*). In: Fifth Workshop on Component-Based Development, Short-Paper, 2005, Juiz de Fora, Brasil.

(Grundy, 2000) Grundy, J. **Storage and retrieval of Software Components using Aspects**. In: 2000 Australasian Computer Science Conference, 2000, Canberra, Australia.

(Guo & Luqi, 2000) Guo, J.;Luqi **A Survey of Software Reuse Repositories**. In: 7th IEEE International Conference and Workshop on the Engineering of Computer Based Systems, 2000, Edinburgh, Scotland.

(Hall, 1999) Hall, P. **Architecture-driven component reuse**. Information and Software Technology, Vol. 41, No. 14,1999, p. 963--948.

(Henninger, 1994) Henninger, S. **Using Iterative Refinement to Find Reusable Software**. IEEE Software, Vol. 11, No. 5, 1994, pp. 48-59.

(Henninger, 1997) Henninger, S. **An Evolutionary Approach to Constructing Effective Software Reuse Repositories**. ACM Transactions on Software Engineering and Methodology, Vol. 06, No. 02, 1997, pp. 111-140.

(Isakowitz & Kauffman, 1996) Isakowitz, T.;Kauffman, R. J. **Supporting Search for Reusable Software Objects**. IEEE Transactions on Software Engineering, Vol. 22, No. 06, 1996.

(Maarek et al., 1991) Maarek, Y. S.;Berry, D. M.;Kaiser, G. E. **An Information Retrieval Approach for Automatically Constructing Software Libraries**. IEEE Transactions on Software Engineering, Vol. 17, No. 08, 1991.

(McIlroy, 1968) McIlroy, M. D. **Mass Produced Software Components**, In: NATO Software Engineering Conference Report, Garmisch, Germany, October, 1968, pp. 79-85.

(Merad & Lemos, 1999) Merad, S.;Lemos, R. D. **Solution Concepts for the Optimal Selection of Software Components**. In: International Workshop on Component-Based Software Engineering, Held in conjunction with the 21st International Conference on Software Engineering (ICSE), 1999, Los Angeles - CA - USA.

(Mili et al., 1995) Mili, H.;Mili, F.;Mili, A. **Reusing Software: Issues and Research Directions**. IEEE Transactions on Software Engineering, Vol. 21, No. 06, 1995, pp. 528-562.

(Mili et al., 1997) Mili, R.;Mili, A.;Mittermeir, R. T. **Storing and Retrieving Software Components : A Refinement Based System**. IEEE Transactions on Software Engineering, Vol. 23, No. 7, 1997.

(Omg, 2002) Omg **CORBA Components**. Object Management Group. 2002.

(Podgurski & Pierce, 1993) Podgurski, A.;Pierce, L. **Retrieving Reusable Software By Sampling Behavior**. ACM Transactions on Software Engineering and Methodology, Vol. 2, No. 03, 1993, pp. 286--303.

(Prieto-Díaz, 1991) Prieto-Díaz, R. **Implementing Faceted Classification for Software Reuse**. Communications of the ACM, Vol. 34, No. 5, 1991.

(Ravichandran & Rothenberger, 2003) Ravichandran, T.;Rothenberger, M. A. **Software Reuse Strategies and Component Markets**. Communications of the ACM, Vol. 46, No. 08, 2003, pp. 109-114.

(Seacord, 1999) Seacord, R. C. **Software Engineering Component Repositories**. In: International Workshop on Component-Based Software Engineering, Held in conjunction with the 21st International Conference on Software Engineering (ICSE), 1999, Los Angeles, CA, USA.

(Seacord et al., 1998) Seacord, R. C.;Hissam, S. A.;Wallnau, K. C. **Agora: A Search Engine for Software Components**. CMU/SEI - Carnegie Mellon University/Software Engineering Institute. 1998.

(Silveira, 2000) Silveira, G. E. D. **Spontaneous Software: A Web-based, Object Computing Paradigm**. In: ICSE 2000 - 22nd International Conference on Software Engineering, 2000, Ireland.

(Szyperski, 1999) Szyperski, C. **Component Software: Beyond Object-Oriented Programming**. Addison Wesley, 1999.

(Traas & Hillegersberg, 2000) Traas, V.;Hillegersberg, J. V. **The Software Component Market on the Internet - Current Status and Conditions for Growth**. ACM SIGSOFT Software Engineering Notes, Vol. 25, No. 01, 2000, p. 114-117.

(Vitharana, 2003) Vitharana, P. **Risks and Challenges of Component-Based Software Development**. Communications of the ACM, Vol. 46, No. 08, 2003, pp. 67-72.

(Vitharana et al., 2003) Vitharana, P.;Zahedi, F.;Jain, H. **Design, Retrieval and Assembly in Component-Based Software Development**. Communications of the ACM, Vol. 46, No. 11, 2003, pp. 97-102.

(Wallnau, 2003) Wallnau, K. C. **A Technology for Predictable Assembly from Certifiable Components**. CMU/SEI - Carnegie Mellon University/Software Engineering Institute. 2003.

(Ye & Fischer, 2002) Ye, Y.;Fischer, G. **Supporting Reuse By Delivering Task-Relevant and Personalized Information**. In: ICSE 2002 - 24th International Conference on Software Engineering, 2002, Orlando, Florida, USA.

(Zhuge, 2000) Zhuge, H. **A problem-oriented and rule-based component repository**. The Journal of Systems and Software, Vol. 50, 2000, pp. 201-208.

Chapter 10
Conclusion

At the same time, this book was a formidable task to complete so far and is an incipient effort towards the ultimate goal of giving full coverage, both academic and industrial, of the field of component reuse in software engineering. Software, on its own, is starting to be noticed as the core of many, maybe most of the industrial, economic and social situations we encounter in our everyday life. It is not unlikely that, in the near future, all forms of organized human life we come across will be, somehow, mediated by software. Family and personal relationships are not bound to escape that -say- trap, we already know; it is not one or only a few million people that have found social networks of great help to developing and improving their personal connections. Behind social networks, of course, there is software. As well as in cars' brakes, plane's autopilots, air traffic control, medical instruments and just about everything, including the electronic lock in my front door.

Producing gazillion lines of code to support such a widespread use of software is a daunting task by all measures. Even more so because, as we practitioners know, a large part of it is repeated here, there and (nearly) everywhere. How many, as an example, USB drivers are there, out there? Nobody knows. But it is certainly much more than the 50.000 instances found by a large code search engine, published on the internet. The cost of writing, time after time and again, these multiple and almost necessarily similar versions of the same basic code cannot be supported for much longer. History shows that every time we face an operational crisis that need a large number of humans to solve, technology -and (or) methods and processes- is introduced to diminish our reliance upon a large pool of almost under used human capital. That was the case of the human operated phone exchange: if technology had not intervened, it would not be possible to carry out all the phone calls that automated (to unthinkable extent nowadays, if viewed from the 50's) exchanges and the internet facilitate and carry nowadays.

We think software has long been facing a similar crisis. There is not enough educated human capital to produce all the software the economy and society need. A picture that is made much worse if we dared to say… all the quality software the economy and…, raising the bar on the quality issue, a situation that could be made even more critical if we added a… within controlled time and cost constraints… condition. Note that the latter has nothing to do with “faster” and “cheaper”, but with controllability and accountability of the process. This is what the themes dealt with in this book are all about, not in the sense of working towards, or with, methods and tools to improve the process of writing more software, but trying to describe possible footpaths leading to the, maybe more achievable, objective of writing less software.

Software reuse, in the form of principles, processes which are reuse centered, focused or influenced, component based development, software reengineering and reverse engineering, metrics, certification, repositories, search and retrieval, as we presented and discussed over the last two hundred or so pages, is in some sense old hat: almost forty years have passed since the NATO 1968 conference where the problem of mass production of such complex knowledge artifacts as software was discussed at large. But this old engineering issue has seen little practical improvement over the previous decades, for a number of reasons, which could be summarized, may be, in the following few bullets:

- Programming, as a primary activity in the development of software, became much more efficient, with the number of executable instructions generated per line of written code increasing by two orders of magnitude since the mid-sixties, due to advances in programming languages, libraries and development environments, among other improvements. That does not mean software became better in any sense (although anecdotal evidence obviously would say so), but that we were able to write much more “software”, as the CPUs see, per programmer, over the same amount of time;
- The process of developing software, on the other hand, went through several waves of improvement, from better team management to leaner

processes, on one hand, to more detailed processes on the other, increasing perceived productivity, making (for example) the “implementation” of “use cases” a phenomenon that has, nowadays, the mass scale of the phone operators of the fifties, whose work, surroundings and result surely could be likened to those of the period’s “industrial” factories. All the same, the software “factories” of today are still an early stage in the life of the software economy: it is unthinkable that we will only achieve the penetration that software will necessarily have in all our ways of life by herding more and more people to implement “cases” and maintain the tens of millions of lines of legacy code (and systems) that large and complex institutions need to keep functioning nowadays;

- Furthermore, the education of software engineers is still in its infancy, with very few institutions really prepared to turn students into software engineers, finishing their degrees with a set of skills capable of delivering real engineering results while writing software -and not simply programs. This educational gap might persist for a number of years and it ought to be treated as a fundamental hurdle in the way of -for example (dear to the authors)- introducing ways and means of writing less software in institutions that either provide or depend upon it.

So, where do we stand now? People are hacking code and is that ok? Is the software life cycle under control? Is it possible to improve the education of software engineers in a way that they would care, and much more, for software in the long run?

The answer to the first question is… maybe… yes. Of course there is a lot of conscious and careful programming around. But there is even more programming aimed and “getting that spec done and away from me”. We have to overcome that before we were able to say that we are only programming what we should be, that we would be taking into account the immense efforts or our (and other) organizations that could be serving as shoulders of giants for our own work. This book is about that.

Software life cycle is… more or less… under control. Given that, in most places, you do not consider the work that has been done before, nor the process of organizing it to be reused (or not, maybe to serve as an example of what not to do), even less so the work that should be done now to prepare all sorts of artifacts that are inherent to the software development process to be useful and used in the future. This book is also about that, about the processes we ought to take into account and perform to make sure we can write more useful and needed software.

Finally, we ought to start considering deep changes in the way we educate software engineers. All engineers are, of course, knowledge workers, people that turn learning and reasoning into artifacts. Software engineers, on top of that, are creators of abstract objects, more often than not virtual worlds that, within computers, create simulacra of the world they deal with, as a way of processing data, out here. Such abstract objects, in the business form of banking automation or an auto pilot, are not mere tools, in the common sense of the word. For the most part, we cannot identify, readily, the complexity of their state spaces, more often infinite than not. We have to change the education of software engineers to take that into account, and one of the best ways to do so, in our opinion, is to convince people to reuse software. This book is about that, it is about identifying what is ready and making use of it again. Because if something is ready and working somewhere, and if we were trained and had the tools, methods and processes to find that, we could care to write only the software that has not been written yet, in the same way that, after publication in a learned journal and understanding by a community, nobody goes around rewriting proofs for classical mathematical theorems.

This book is a tiny step in the direction of showing what has been achieved so far in the software reuse field. We did not mean to cover the subject in its fullness, neither to be the authoritative source on component reuse in software engineering. We hope we did contribute, by writing these pages, to introduce the subject to more people, to raise the level of awareness both to the benefits of software reuse and the challenges of getting there.

Finally, this book is bound to be in a beta state, as it is common these days, for quite a few moons. There are errors, misconceptions and misinterpretations, there are things we now know we should have included and we did not and, maybe worse, there are things we even now do not know we would have to have included. To fix these shortcomings, our only hope is that, being this an open work, our effort is just a starting point to something bigger, better, more accurate and pedagogical that the community that -if we are lucky- is built around this seed will most certainly build.

So far, it took us a lot of time and it has been a pleasure to try and put together something that could be of help to others and value to some. To all of you that helped us to get to this point, many thanks. To those of you that want to help somehow, please do. We are always going to be open…

The Authors

Eduardo Santana de Almeida works at the Recife Center for Advanced Studies and Systems (C.E.S.A.R.) as a reuse consultant and project manager. He received his BSc. in computer science in 2000 from the Salvador University (UNIFACS) and his M.Sc. in computer science in 2003 from the Federal University of São Carlos (UFSCar). He is involved in four industrial projects that focus on different aspects of reuse, such as processes, environments and tools. Eduardo Almeida is (co-)author of over 60 referenced publications presented at conferences such as WCRE, IRI, ECOOP, and EUROMICRO, amongst others. Moreover, he is the leader of the RiSE - Reuse in Software Engineering - group and is concluding his Ph.D. in computer science at the Federal University of Pernambuco. His research areas include processes, methods, environments and tools related to software reuse.

Contact: eduardo.almeida@cesar.org.br

Alexandre Alvaro is a senior member of the RiSE (Reuse in Software Engineering) group, at the Federal University of Pernambuco. He received his M.Sc. in computer science from Federal University of Pernambuco in 2005 and he is a Ph.D. candidate in Computer Science at the Federal University of Pernambuco. Nowadays, he is a Systems Engineer at the Recife Center for Advanced Studies and Systems (C.E.S.A.R.). Author of several papers in important vehicles, such as ECOOP and Euromicro conferences, in software reuse, component-development and software component certification areas, he has presented research works at conferences in Brazil and several other countries, including Portugal, United States and Hong Kong. He is involved in four industrial projects that focus on different aspects of reuse, such as processes, environments and tools.

Contact: alexandre.alvaro@cesar.org.br

Vinicius Cardoso Garcia is an expert member of the Brazilian Computer Society, and a member of the RiSE - Reuse in Software Engineering – Group, at the Federal University of Pernambuco. He received his BSc. in computer science from the Salvador University (UNIFACS) in 2001, and the M.Sc. in computer science from Federal University of São Carlos in 2005. He is currently a Ph.D. candidate at the Federal University of Pernambuco since 2005, and also a Systems Engineer at the Recife Center for Advanced Studies and Systems (C.E.S.A.R.) since 2005. Vinicius Garcia is (co-)author of over 30 referenced publications presented at conferences such as WCRE, IRI, ECOOP, CBSE, ICSR and EUROMICRO, amongst others. He is currently involved in four research projects in the computer Science area, more specifically in Software Reuse Maturity Models and Software Reuse Adoption in Software Development Process.

Contact: vinicius.garcia@cesar.org.br

Jorge Cláudio Cordeiro Pires Mascena, Software Engineering Supervisor at Modular Mining Systems Inc. since 2006, received his M.Sc. in Computer Science at the Federal University of Pernambuco in 2006. Among other important duties in Modular, he is the Chairperson of the Software Development Working Group, with approximately 30 members. This group aims at improving software development practices and reuse activity in Modular. He has worked as Software Architect at C.E.S.A.R (Recife Center for Advanced Studies and Systems) from 1997 to 2006, where he had crucial participation in the development of several large-scale, mission-critical systems, both in Brazil and in the United States. He was also a member of the C.E.S.A.R reuse program committee from 2001 to 2006, with special emphasis on reuse metrics.

Contact: jorge.mascena@gmail.com

Vanilson André de Arruda Burégio has been working at the Information Technology Agency of Pernambuco (ATI) as a Systems Analyst, where he has been involved in the development of the Pernambuco Government Financial System (E-FISCO). He worked at the Recife Center for Advanced Studies and Systems (C.E.S.A.R.) as a Systems Engineer since 2001 until 2006, where participated in the development of several large-scale information systems and was engaged as a team leader of two industrial reuse projects with a focus on environments and tools to support the reuse process. Vanilson Burégio is a Sun Java Certified Programmer, Sun Java Certified Web Component Developer. He has been participating in the development of diverse Java-based systems over the last seven years. He received his M.Sc. in Computer Science at the Federal University of Pernambuco in 2006. His research areas are software architecture, software quality, software reuse environments and tools.

Contact: vanilson@gmail.com

Leandro Nascimento is a member of the RiSE - Reuse in Software Engineering – Group, and is a Bachelor of Computer Science at Federal University of Pernambuco. He is a M.Sc. Candidate in Computer Science at Federal University of Pernambuco. As the result of his graduation work, he presented a Framework for Graphical User Interface development in J2ME. Nowadays, he is a System Engineer at the Recife Center for Advanced Studies and Systems (C.E.S.A.R) and has been working on J2ME projects, developing research into the area related to Component-Based Software

Contact: leandro.nascimento@cesar.org.br

Daniel Lucrédio graduated and got his M.Sc. degree in Computer Science at the Federal University of São Carlos, and is currently a Ph.D. candidate at the University of São Paulo – São Carlos. He is the author of MVCASE, a free tool for UML modeling and component-based development, three times awarded at the Brazilian Symposium on Software Engineering, in 2001, 2002 and 2004. Author of several papers in important vehicles, such as IEEE and Euromicro conferences, in the software reuse and component-development areas, he presented research works and speeches at conferences in Brazil and several other countries, including

Canada, United States, France and Hong Kong. He works as a software development coordinator, and is one of the creators of São Carlos' JUG - Java Users Group, which researches and disseminates the most recent technologies used in the industry.

Contact: lucredio@icmc.usp.br

Silvio Lemos Meira is the chair of Software Engineering at the Center for Informatics at the Federal University of Pernambuco in Recife, Brazil. Prof. Meira holds a B.Sc. in Electronic Engineering from ITA, S. J. Campos, Brazil, a M.Sc. in Computer Science from the Federal University of Pernambuco and a Ph.D. in Computing, under David Turner, from the University of Kent at Canterbury, UK. Among many other responsibilities, Prof. Meira is the leader of the Reuse in Software Engineering (RiSE) group and the Chief Scientist at C.E.S.A.R. (the Recife Center for Advanced Studies and Systems). Silvio Lemos Meira is (co-) author of well over 150 referenced publications in software engineering conferences and journals.

Contact: silvio.meira@cesar.org.br

This book addresses practitioners, software reuse researchers, professors and students interested in the topic of software reuse. The authors describe, in detail, the essential foundations, principles, techniques and tools related to software reuse.

The authors are professionals and researchers with significant knowledge of the topic and have successfully applied software reuse principles within industry environments. They have structured the book around a comprehensive framework which is being applied in industry and universities in order to teach software reuse concepts.

As a student, you will find a detailed roadmap of the key processes, their activities, and their strong and weak points, timelines, and underlying directions for the exploration of software reuse. As a researcher or lecturer, you will find a comprehensive state-of- art discussion organized into a comprehensive and detailed framework. As a professional, you will discover the techniques, their current state, their advantages and drawbacks for introducing a reuse program based on education, processes, measurements and tools.

RiSE

C.E.S.A.R

C.E.S.A.R e-books