Pós-Graduação em Ciência da Computação

RiDE: The RiSE Process for Domain Engineering

by

Eduardo Santana de Almeida

Ph.D. Thesis in Computer Science

RECIFE, MAY/2007

UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA

Eduardo Santana de Almeida

RiDE: The RiSE Process for Domain Engineering

Thesis presented to the Graduate Program in Computer Science of the Universidade Federal de Pernambuco in partial fulfillment of the requirements for the degree of Doctor of Science.

Advisor: Prof. Silvio Romero de Lemos Meira

"..... você não sabe o quanto eu caminhei, pra chegar até aqui....."

A Estrada, Cidade Negra, 1998

This work is dedicated to God, my wonderful Mom, and my lovely brothers,
Marcelo and Nando.

cknowledgements

A long time ago, William Shakespeare, the foremost poet of the English verse, in Troilus and Cressida, Act 3, Scene 2, said: "*Words pay no debts*". This quote expresses my feeling in this moment and it is impossible to describe all my gratitude to many people who contributed in indescribable and different ways for this so desired moment. Nevertheless, I can incur in the mistake of not thanking all of them. For those, I express my profound apologies.

Initially, I would like to thank all the professors from the Universidade Salvador (UNIFACS) and Universidade Federal de São Carlos (UFSCar), especially, Antonio Atta, Antonio Francisco do Prado and Luis Carlos Trevelin for introducing me in the research world. Perhaps, without the first steps, I could not start this climbing.

Next, I would like to thank professor Gledson Elias from the Universidade Federal da Paraíba (UFPB) for introducing me to my future advisor during this Ph.D. degree and professors Ana Carolina Salgado and Carlos Ferraz from the Universidade Federal de Pernambuco (UFPE) for believing in my potential to be a Ph.D. student at this university.

My gratitude to all the employees at UFPE, from the receptionists, the support team, and secretaries who always had a *lot of patience* with me, up to and including all other M.Sc. and Ph.D. students in my time at UFPE. This university is a great place to do a good job.

The Recife Center for Advanced Studies and Systems (C.E.S.A.R) offered me a perfect and challenging environment to identify, formalize, confront, experiment and test real problems related to several aspects of software reuse through its infra-structure and discussions with experts in different areas. At

this institute, I would like to thank all areas, including support, marketing, acquisition, engineering, up to the top-management. Additionally, this institute was the *first financial support* for my ideas during my period in Recife and without it the results could not be the same. I hope that in a near future, C.E.S.A.R can do the same with many researchers.

This work was also partially supported by the Coordenação de Aperfeicoamento de Pessoal de Nível Superior (CAPES).

Some key researchers in the software reuse area offered valuable feedback for this work. In Chapter 2, Robert Glass and, especially, Jeff Poulin spent important time discussing the key developments in the field of software reuse. Jeff was also a great partner, always discussing with our group other important aspects related to reuse. It is really great to see important researchers spending their time with our ideas trying to improve the field as a whole.

Other key researchers also played a crucial role, receiving me in their universities and institutes for discussing software reuse. Among them, I would like to thank Bill Frakes, Dirk Muthig, and Maurizio Morisio. Additionally, I would like to thank Bertrand Meyer and Colin Atkinson for receiving me as a visiting researcher in their groups and offering all the conditions for doing a good job. This was of great help to the conclusion of this work.

Professors André Santos, Augusto Sampaio, Hermano Moura, from the Universidade Federal de Pernambuco (UFPE), Jones Albuquerque from the Universidade Federal Rural de Pernambuco (UFRPE) and Claudia Werner, contributed with valuable feedback with different points of view during my thesis.

The results of this thesis could not be achieved without the support of the Reuse in Software Engineering (RiSE) group. This group created a great atmosphere to discuss several aspects of software reuse. I do not have doubts that in a near future this group will be one of the top software reuse groups in the world. My *gratitude to all the RiSE members* for their patience in attending seminars during the weekends, giving and listening to internal workshops, for the stimulating discussions on several aspects of software reuse, and for refuting several of my ideas in an elegant way. This is also the job of a *great* group and I am proud of being part of this adventure.

All the templates used in this process were defined by Ana Paula Cavalcanti. Her job as a Software Quality Engineer was incredible in this sense.

For evaluating this work, the experimental study was a hard task to perform and, during this process, many people were important. Initially, I would like to thank all the students who participated in the study, thank you for your dedication and patience. Alexandre Martins and Liana Barachisio were also important in helping me to test the tools, analyze the metrics and configure the entire environment. Guilherme Travassos from the Universidade Federal do Rio de Janeiro (UFRJ) and Márcio de Oliveira Barros from the Universidade Federal do Estado do Rio de Janeiro (Uniriotec) were also important in discussing some aspects related to the study. At the end, *certainly*, the definition, planning and presentation of the study would not be possible without the support of Daniela Cruzes from the Universidade Salvador (UNIFACS).

Additionally, during this period, some close colleagues were also very important. Alexandre Alvaro and Daniel Lucrédio contributed and are still contributing in many discussions related to software reuse, even when we are geographically separated. Lucrédio was also the main reviewer of this work and his patience and attention was immensurable.

Vinicius Garcia was (and will be) a good professional and friend that contributed a lot during this work. Vinicius saw all the dreams start and has been fundamental in this road. He knows that nine years are not nine days. So, thank you very much, Cardoso. Your support in discussions, different points of views, travels, problems ..... was/is very important.

Silvio Meira my guru, friend, and professional *example* – was a strong inspiration and motivation during this entire road. His motivation and dedication to the future of research/industry/city/country is incredible. I strongly admire his legacy and hope doing half a history like his in the future. Silvio was also important by showing me that brilliant ideas can happen even using a piece of paper, by *opening my mind about the relationship between the university and industry* and by showing me that we can do a good job in any place around the world. Silvio, thanks a lot for all the good and bad discussions in your office, house or even in bars; for all the moments and patience to tell and re-tell me several histories about important facts in your career; for opening the

doors and for all the support. Thank you for believing in my potential. I would like to say that working with you during this time was an incredible pleasure.

Loana Medeiros Silva was also very important during my career since the graduate degree. Her support and understanding were significant in the transition from Salvador to Recife and in part of my career and sure I could not forget it, never.

Living away from home is not an easy task, especially when our roots are from Salvador, Bahia and in this period, several people facilitated it with funny and unforgettable moments. Thus, I would like to register my thanks to Cassio Freire, my first roommate in Recife. Those moments with you were very good. Unfortunately, you had to look for your road in another city. Good Luck, Kersin. Cinthya Roberta, this period with you was incredible. I do not have any words to say to you, just: thank you, so much for all the moments which we enjoyed. We know that this result has also your participation which was started since the high school degree. Moreover, the other roomates such as Alexandre Alvaro, Ricardo Argenton Ramos, Ricardo Alexandre Afonso, Vinicius Cardoso Garcia and all the different people, from different places around the world who passed by our place were also important. In this house, our last roommate, Leobino Nascimento Sampaio, was also important for contributing with good moments and several discussions about the future in our hometown.

Manuela Barbosa da Silveira my little love suddenly appeared during the end of this thesis and is being very important in this process. Her caress, understanding, patience-impatience and love have played a key role in my life during this period. I think that this time was also essential for our maturity and to learn a little bit more about each other. Thus, I would like to thank you and your entire family for receiving and taking care of me during it.

My little mom, even so far away, thank you for your existence in my life. I hope to have more time in the future to visit and enjoy life with you.

Finally, *I would like to say that even Shakespeare's verses are insufficient to describe my gratitude to my family*; however, I have to try to do it. My brothers Marcelo and Nando, you were/are and will be unforgettable to me. Even so far away, it is impossible to forget all our great moments. You are always my brothers, fathers, friends and constant inspiration. During all my

moments, good and bad, I never forgot you. Again, I would like to thank my brother Marcelo who during part of his life invested in me, instead of enjoying life, one more time; I do hope to make sure, forever in the future that you were right then. Thank you, brothers. You are very special to me.

Mom, oh mom! I would like to say that you are incredible and everything in my life. Thank you so much, for your love and care. I admire all your efforts to support me and my brothers and this result is dedicated to you and them. I hope to come back home soon in order to enjoy our time together. I love you and all.

At the end, I would like to thank God, Senhor do Bonfim and all the Orixás who protected and guided me during this journey. Axé!!!

Eduardo Santana de Almeida

Recife, Pernambuco, Brazil

April 9, 2007

esumo

A reutilização de software – o processo de criar sistemas através de artefatos existentes, ao invés de desenvolvê-los do zero – é um aspecto chave para melhorias em qualidade e produtividade no desenvolvimento de software. Qualidade pode ser melhorada por reutilizar todas as formas de experiência, incluindo produtos e processos. Por outro lado, a produtividade pode ser aumentada por utilizar experiências existentes. Entretanto, esse processo é mais efetivo quando sistematicamente planejado e gerenciado no contexto de um domínio especifico, onde as aplicações compartilham funcionalidades comuns.

Neste cenário, a Engenharia de Domínio – a atividade de coletar, organizar e armazenar experiências anteriores na construção de sistemas ou partes de sistemas, de um domínio particular, na forma de artefatos reutilizáveis – tem sido vista como um facilitador para obter os benefícios desejados. No entanto, os processos existentes de engenharia de domínio apresentam problemas cruciais, como, por exemplo: eles não cobrem os três passos da engenharia de domínio, a saber, análise do domínio, projeto do domínio e implementação do domínio; além de não definir de forma sistemática as atividades, as sub-atividades, os papéis, as entradas e as saídas de cada passo.

Assim, este trabalho define um processo sistemático para realizar a engenharia de domínio, baseado no estado da arte da área, incluindo os passos de análise, projeto e implementação do domínio. Essa definição foi embasada por *surveys* detalhados sobre reutilização de software e processos de reutilização, cobrindo pesquisas informais, estudos empíricos e relatos de empresas. Esta tese primeiro apresenta os resultados desses *surveys* e, em seguida, descreve o processo proposto discutindo suas atividades, sub-atividades, entradas, saídas, princípios, *guidelines* e papéis. Por fim, são discutidos os resultados de um estudo experimental para análise da viabilidade do processo proposto em um projeto de engenharia de domínio.

Palavras-chave: reutilização de software, engenharia de domínio, análise de domínio, projeto do domínio, implementação do domínio, estudo experimental.

bstract

Software reuse – the process of creating software systems from existing software rather than building them from scratch - is a key aspect for improving quality and productivity in the software development. Quality can be improved by reusing all forms of proven experience, including products and processes, as well as quality and productivity models. Productivity can increase by using existing experience, rather than creating everything from scratch. However, this process is more effective when systematically planned and managed in the context of a specific domain, where application families share some functionality.

In this scenario, Domain Engineering (DE) - the activity of collecting, organizing, and storing past experience in building systems or parts of systems from a particular domain in the form of reusable assets – has been seen as a facilitator to obtain the desired benefits. Nevertheless, the existing domain engineering processes present crucial problems, such as: they do not cover the three steps of domain engineering, for instance, domain analysis, domain design, and domain implementation; besides, they do not define activities, sub-activities, roles, inputs, outputs of each step in a systematic way.

This work defines a systematic process to perform domain engineering based on the state-of-the-art of the area, which includes the steps of domain analysis, domain design, and domain implementation. This definition was based on extensive surveys on the software reuse and reuse processes areas, covering academic and industrial studies, papers and reports. This thesis first presents the results of these surveys, and then presents the proposed process, discussing it in terms of its activities, sub-activities, inputs, outputs, principles, guidelines and roles. Finally, it discusses the findings of an experimental study in order to analyze the process applicability in a domain engineering project.

Keywords: software reuse, domain engineering, domain analysis, domain design, domain implementation, experimental study.

Table of Contents

ist of Figures

ist of Tables

ist of Acronyms

Introduction

"The journey of a thousand miles begins with a single step"

Lao Tzu (604 BC – 531 BC)
Chinese philosopher

In general, the reuse of products, processes and other knowledge can be important ingredients to try to enable the software industry to achieve the pursued improvement in productivity and quality required to satisfy growing demands. However, these efforts are often related to individuals and small groups, who practice it in an ad-hoc way, with high risks that can compromise future initiatives in this direction. Currently, organizations are changing this vision, and software reuse starts to appear in their business agendas as a systematic and managerial process, focused on application domains, based on repeatable and controlled processes, and concerned with large-scale reuse, from analysis and specification to code and documentation.

Thus, systematic software reuse is a paradigm shift in software development from building single systems to application families of similar systems (Frakes & Isoda, 1994).

The instantiation of this problem is the main subject of this thesis, which will discuss it in detail, beginning with the problem formulation, passing through the existing solutions, ending with the current proposal. This Chapter contextualizes the focus of this work and starts by presenting its motivation and a clearer definition of the problem.

1.1. Motivation

Software reuse – the process of using existing software artifacts rather than building them from scratch (Krueger, 1992) – is generally regarded as the most important mechanism for achieving software development more efficiently. This vision started to be idealized in the end of 1968, when McIlroy (1968), motivated by the software crisis, wrote a seminal paper on software reuse entitled "*Mass Produced Software Components*". Since that time, many discussions took place, involving issues ranging from the possibility of a software industrial revolution (Cox, 1990), in which programmers would stop coding everything from scratch and begin assembling applications from well-stocked catalogs of reusable software components, to silver bullets (Moore, 2001) based on software reuse.

In the software development field, many works discuss the reuse impact on improvements on software quality, as well as on costs and productivity (Lim, 1994), (Mili et al., 1995), (Basili et al., 1996), (Sametinger, 1997), (Frakes & Succi, 2001), (Ezran et al., 2002), (Poulin, 2006). On the other hand, other fields such as psychology have long argued that humans seldom solve problems from first principles (Mili et al., 2002). Thus, when faced with a problem to solve, initially a "rote recall" is performed, just in case this problem was already solved. Next, when that fails, an "approximate recall" follows, with the hope of identifying an already solved problem that is so close that its solution can be adapted, locally, to address the new problem. Finally, only when that fails also, people fall back to analytic problem solving, at least as far as decomposing the initial problem into a set of more manageable sub-problems.

Nevertheless, in the software development field - the focus of this thesis - studies showing these benefits, in most cases, are restricted to small and academic contexts, and there is a general notion that such findings could somehow be extrapolated to more generic contexts. However, as reports from large software development organizations (Bauer, 1993), (Endres, 1993), (Joss, 1994), (Griss, 1994), (Griss, 1995) and as empirical and informal research indicate (Card & Comer, 1994), (Rine (1997), (Glass, 1998), in fact this notion is still far from true.

Many reasons have been enumerated (Mili et al., 1999), (Poulin, 1999), (Moore, 2001), (Morisio et al., 2002) to justify the inability of the software development community to fully accomplish a rather simple and universal concept, at least in theory (Mascena, 2006). The practice, however, has proved to be a rather complex task, which poses a number of myths (Frakes & Fox, 1995), inhibitors (Bass et al., 2000), facts and fallacies (Glass, 2002), besides technical and non-technical challenges.

In this context of growing challenges, the Reuse in Software Engineering (RiSE) project[1] was proposed (Almeida et al., 2004). RiSE's goal is to develop a robust framework for software reuse in order to enable the adoption of a reuse program. The proposed framework has two layers, as shows Figure 1.1. The first layer (on the left side) is formed of best practices related to software reuse. Non-technical aspects, such as education, training, incentives, program to introduce reuse, and organizational management are considered. This layer constitutes a fundamental step before the introduction of the framework in organizations. The second layer (on the right side), is formed of important technical aspects related to software reuse, such as processes, environment, and tools.

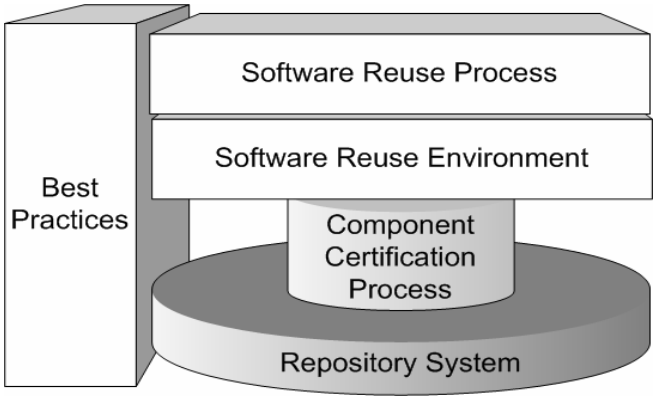

Figure 1.1. The RiSE Framework for Software Reuse.

As can be seen in Figure 1.1, the RiSE project addresses reuse aspects not included in the scope of this thesis, such as software reuse environments (Garcia et al., 2006a), component managers (Burégio, 2006), component certification (Alvaro et al., 2006), besides other tools proposed by the project, including

[1] The RiSE project in the web: www.rise.com.br

domain analysis tools and component search engines (Garcia et al., 2006), (Mascena et al., 2006a), (Vanderlei et al., 2006).

The reuse process in which this thesis is inserted is composed by several requirements (Chapter 3 discusses these requirements in details), such as: domain engineering, application engineering, metrics, economics, reengineering, adaptation, and quality, which constitute other ongoing research. These efforts are coordinated and will be integrated in an industrial scale reuse solution.

All these requirements are important and relevant for a reuse process. However, in this thesis the focus is on Domain Engineering, which is considered a key aspect to success in reuse, especially when considering software reuse processes.

1.2. Problem Statement

A process can be understood as a collection of related tasks leading to a product (Ezran et al., 2002), or simply, a way of defining specifically who does what, when, and how (Fayad, 1997). A process is important to define how an organization is supposed to perform its activities, and how people work and interact, in order to ensure efficiency, reproducibility, and homogeneity (Ezran et al., 2002). Several works discuss the benefit and improvements of using a well-defined software process (Basili et al., 1995), (McConnell, 1998).

In the context of software reuse, important research *including company reports* (Bauer, 1993), (Endres, 1993), (Griss, 1994), (Joos, 1994), (Griss, 1995), *informal research* (Frakes & Isoda, 1995), (Frakes & Kang, 2005) and *empirical studies* (Rine, 1997), (Morisio et al., 2002), (Rothenberger et al., 2003) have highlighted the relevance of a reuse process, since the most common way of software reuse involves developing applications reusing pre-defined assets.

The software reuse processes literature focuses on two directions: **Domain Engineering** and **Software Product Lines**. Although the software reuse community does not make a clear distinction between the terms (in Chapter 3 a more detailed discussion is presented), this thesis focuses on domain engineering, a consensus between researchers from both communities.

However, existing reuse processes present crucial problems, such as not covering the three classical steps of domain engineering, domain analysis, design and implementation; besides, they do not define activities, sub-activities, roles, inputs, outputs of each step in a systematic way.

Thus, the goal of the work described in this thesis can be stated as:

> *This work defines a systematic software reuse process to perform domain engineering, which includes the steps of domain analysis, domain design and domain implementation, based on a set of activities, sub-activities, inputs, outputs, principles, guidelines and roles. Moreover, the process is based on the state-of-the-art in the area and its foundations and elements are discussed in details.*

1.3. Overview of the Proposed Solution

In order to achieve this goal, stated in the previous Section, the RiSE process for Domain Engineering is proposed. The process is based on the following foundations[2]:

- **Process Model.** The domain engineering process is based on the spiral process model (Boehm, 1988); however, it also presents some characteristics of the Component-Based Software Engineering (CBSE) process model (Pressman, 2005), since reusable assets are used to develop applications**;**
- **Domain Driven.** The process is domain-driven, with focus on a set of applications from a particular domain, instead of a single application**;**
- **Iterative and Incremental.** The process is iterative and incremental, since in domain engineering projects, which are based on a set of applications in a domain, it is important to divide the effort in iterations and increments in order to more easily manage the activities; and
- **Well-defined software engineering concepts.** The process is based on important software engineering concepts that can increase the reuse potential, such as software architecture, component-based development and design patterns.

[2] In chapter 4, all these aspects are discussed in detail.

Besides these foundations, the domain engineering process is composed of three steps: *domain analysis*, *domain design*, and *domain implementation*. In summary, domain analysis identifies, organizes, and documents the common and variable features of applications in a domain; domain design aims at designing a domain-specific software architecture that can support several applications in that domain; and, finally, domain implementation aims at implementing and documenting the reusable assets, in order to be reused by applications in a domain.

1.4. Out of Scope

As the proposed reuse process is part of a broader context, a set of related aspects will be left out of its scope. Nevertheless, as these aspects were envisioned since the initial definitions of the process, they can be added in the future with some adjustments. Thus, the following issues are not directly addressed by this work:

- **Application Engineering.** An important issue in a reuse process is to define the development *with* reuse activity, which consists in developing applications reusing the assets produced in domain engineering. However, this aspect can be as complex as domain engineering, involving the definition of activities, sub-activities, inputs, outputs, and roles;
- **Metrics.** Measurement activities are essential in any engineering process. The software reuse metrics field is a wide road that encompasses aspects related to benefits in productivity and reusability obtained with a domain engineering effort, for example. However, even with some metrics being used in domain scoping and domain design, there are more specific reuse metrics (Poulin, 1997) that could be incorporated into the process;
- **Economics.** Business aspects are an important instrument to aid managers to start a domain engineering project, since it can show the return on investment in a specific domain. Nevertheless, it is not a trivial issue. This can be verified, for example, by the fact that no reuse process has a way of addressing it;

- **Reengineering.** A difficult and often necessary issue for an organization is to reengineer systems to be reused, since most organizations develop either have developed similar systems in the same domain without any presence of future reuse concern. However, due to its complexity, this can be seen as a sub-area or discipline of software engineering, and there are no efforts to integrate reengineering aspects in software reuse processes, even it being important. Krueger (2002) has presented some directions in this area, but not connected to a reuse process;
- **Adaptation.** It is known that a well-defined reuse process can bring benefits for the organizations that adopt it. However, currently, several organizations have their own software development processes in which the roles and the activities are defined and coordinated, at least, in theory. Thus, in order to decrease the impacts and risks, it is important for a reuse process to define how to do the tailoring for an existing organizational process. Although some empirical studies show this adaptation as a critical point in a reuse project (Morisio et al., 2002), this issue is not considered in reuse processes, which can present barriers to organizations interested in adopting it; and
- **Quality.** The last issue is related to the quality of the reusable assets. It is important because since the end of 1980s (Bauer, 1993), organizations develop and manage component libraries or repository systems. Thus, if a developer reuses an asset with defects, this may be a future inhibitor for reuse. However, the available reuse processes do not discuss, for example, ways to test, inspect and certify the produced assets. There are directions in this sense (Meyer et al., 1998), but normally not related to a reuse process.

1.5. Statement of the Contributions

As a result of the work presented in this thesis, the following contributions can be enumerated:

- An extensive study of the key developments in the field of software reuse, in an attempt to analyze this research area and identify trends to follow (Almeida et al., 2007);
- A survey of the state-of-the-art of software reuse processes in order to understand and identify the weak and strong points of existing processes (Almeida et al., 2005a);
- The formalization and definition of the domain engineering process presented in this thesis (Almeida & Meira, 2005);
- The definition and systematization of the steps of the domain engineering process detailing the approaches for domain analysis (Almeida et al., 2006), domain design (Almeida et al., 2007a), and domain implementation (Almeida et al., 2007b); and
- The definition, planning, operation, analysis, interpretation, and packaging of an experimental study which evaluates the viability of the proposed process for domain engineering (Almeida et al., 2007c).

1.6. Organization of the Thesis

The remainder of this thesis is organized as follows.

Chapter 2 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.

Chapter 3 reviews fifteen software reuse processes representing the state-of-the-art of the area and discusses the requirements and important issues that systematic reuse processes must consider.

Chapter 4 presents an overview of the proposed domain engineering process, its foundations, elements, and steps.

Chapter 5 describes the domain analysis approach, which is the first step of the domain engineering process, its principles, activities, sub-activities, and roles.

Chapter 6 presents the domain design approach, which is the second step of the domain engineering process, its principles, guidelines, activities, sub-activities, and roles.

Chapter 7 describes the domain implementation approach, which is the third and last step of the domain engineering process, its principles, activities, sub-activities, and roles.

Chapter 8 presents the definition, planning, operation, analysis, interpretation, and packaging of the experimental study which evaluates the viability of the proposed process.

Chapter 9 summarizes the contributions of this work, presents the related work, and directions for future work.

Appendix A describes the recommended references on several aspects of software development and software reuse.

Appendix B shows graphically the reference distribution included in this thesis.

Appendix C presents the template for the scope definition, which is part of the domain analysis step.

Appendix D contains the template for domain documentation, describing in details the planning, the applications, the features and the domain.

Appendix E presents the template for documenting the assets produced during the *group component* sub-activity.

Appendix F contains the template responsible for representing the component specification defined in Domain Design step. For each component, it presents a description, use cases, features, workflow, class diagrams, interfaces, packaging, and quality attributes.

Appendix G describes the template responsible for documenting the domain architecture defined in the Domain Design step, presenting its goals, modules, quality attributes, views, and component diagrams.

Appendix H presents the questionnaires used in the experimental study.

Figure 1.2 represents the roadmap to the thesis. The references and appendixes were omitted to improve the legibility.

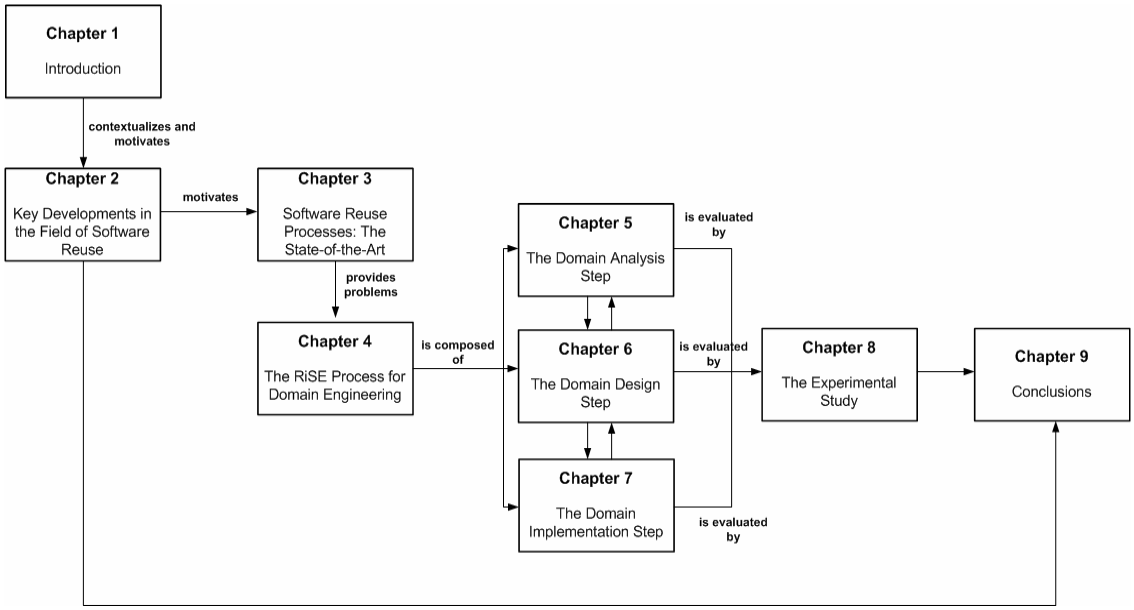

Figure 1.2. Roadmap to the Thesis.

Key Developments in Software Reuse

"Study the past if you would define the future"

Confucius (551 BC – 479 BC)
Chinese philosopher and reformer

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 innovation that have pushed forward the frontiers of techniques to improve software development processes. From subroutines in the 1960s through modules in the 1970s, objects in the 1980s, and components in the 1990 (Clements & Northrop, 2001), 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, among others (Broy, 2006).

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 (Almeida et al., 2007).

2.1. Introduction

Code scavenging, reengineering, code generators or case tools can contribute to increased productivity in software development. However, 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.

The next Sections discuss motivations and definitions for software reuse.

2.2. Motivation

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 (Cox, 1990), (Ran, 1999). 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 (OOP), Component-Based Development (CBD), Domain Engineering (DE), and Software Product Lines (SPL), among others. These advances are known ways to achieve software reuse.

In the software reuse and software engineering literature, there are different published rates about reuse (Poulin, 2006), however, some studies 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 explained next (Basili et al., 1996).

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 (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 thesis, 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 the 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 current state-of-the-practice, we can cite: 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, the lack of Software Reuse Assurance processes before publishing artifacts for reuse, among other aspects that will be discussed in the following Sections. 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 library and repository concepts. Mili et al. (1998) discuss about 50 approaches for this problem. Prieto-Diaz & Freeman's (1987) work has given an important contribution to this area.

Prieto-Diaz & Freeman's 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 results 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 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 asked reusers to rank functionally equivalent components. Next, they 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.

Currently, facets are still used in reuse tools (Garcia et al., 2006) in order to obtain the presented benefits.

2.6. The Benefits

As discussed previously, software reuse has a positive impact on software quality, as well as on cost and productivity (Lim, 1994), (Basili et al., 1996), (Sametinger, 1997), (Frakes & Succi, 2001).

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 than would be the case for a component that is developed and used only once;
- **Productivity**. A productivity gain is achieved due to less code that has to be 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 increases 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 assets 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 assets can result in a reduction of that time;
- **Documentation**. Although documentation is very important for the maintenance of a system (Lethbridge et al., 2003), 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.

Ezran et al. (2002) presented some important reported estimates[1] of actual improvements due to reuse in organizations using programming languages ranging from Ada to Cobol and C++:

- **DEC**
  - *Cycle time*: 67%-80% lower (reuse levels 50-80%)
- **First National Bank of Chicago**
  - *Cycle time*: 67%-80% lower (reuse levels 50-80%)

[1] However, how these data were measured and what languages were used in each company is not discussed.

- **Fujitsu**
  - *Proportion of projects on schedule*: increased from 20% to 70%
  - *Effort to customize package*: reduced from 30 person-months to 4 person-days
- **GTE**
  - *Cost*: $14M[2] lower (reuse level 14%; baseline costs not specified)
- **Hewlett-Packard**
  - *Defects*: 24% and 76% lower (two projects)
  - *Productivity*: 40% and 57% higher (same two projects)
  - *Time-to-market*: 42% lower (one of the above two projects)
- **NEC**
  - *Productivity*: 6.7 times higher
  - *Quality*: 2.8 times better
- **Raytheon**
  - *Productivity*: 50% higher (reuse level 60%)
- **Toshiba**
  - *Defects*: 20%-30% lower (reuse level 60%)

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,

[2] All the values presented in this thesis are in U.S. dollars.

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 that is not for free. Cost associated with reuse can be (Poulin, 1997), (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 (among others things) and archives, with payoffs being realized only years later. Developing assets for reuse is more expensive than developing them for single use only (Poulin, 1997). 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 retrieval of 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 retrieve them. Moreover, it is important to have a well-organized repository containing components with some means of accessing it (Mili et al. 1998), (Lucrédio et al. 2004).
- **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 systems, 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 the developer wants. 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 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, this non-systematic reuse will probably 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 whole software process, and into the software process improvement program;
- ensuring all software staff have the necessary competence and motivation;
- establishing appropriate organizational, technical and 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 to give full visibility to the flow, 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 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 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 practice. In this Section, the three main cases found in the literature - IBM, Hewlett-Packard (HP) and Motorola - are presented and discussed.

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 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 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 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 at 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 organizations.

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 his 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* (CMM)*".*

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 may be necessary, 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 considered.

With the maturity of the corporate reuse program, Griss has developed an interesting incremental model for reuse adoption. According to this model, 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 processes. 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 in the early 90's, had software reuse as one of its pillars (Joss, 1994). One of the first problems to be dealt with was to change the software development environment from a hardware to a software mindset. When 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 retrieve 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 of 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 were 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 responsibility 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. Even 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 substantial 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, since 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 toolkit was scheduled for a prototype in the first quarter of 1996, however, the literature does not present any results related to it.

Other examples of success in software reuse may be seen on the annual hall of fame in software product lines, where the most distinguished efforts related to software product lines are elected during the Software Product Line Conference (SPLC).

The following excerpt from SEI's website[3] describes the rationale of this effort: "*Each Software Product Line Conference (SPLC) culminates with a session in which members of the audience nominate systems for induction into the Software Product Line Hall of Fame. Those nominations fed discussions*

[3] http://www.sei.cmu.edu/productlines/plp_hof.html

about what constitutes excellence and success in product lines. The goal is to improve software product line practice by identifying the best examples in the field. Nominations are voted on at the next SPLC by the majority of those present. For example, the Bosch Gasoline Systems: Engine Control Software Product Line and Philips Low-end Television Product Line were nominated at SPLC 2006. The Bosch Gasoline Systems: Engine Control Software Product Line was inducted at SPLC 2007."

Also relevant to the new industrial tendencies regarding software reuse is the appearance of companies created with a specific focus on software reuse. With solutions dedicated to help in different aspects of software development by using some reuse-related technique, these companies normally have reuse-related products and tools in their portfolio, and offer consulting and training services in software reuse. Most of these companies were founded by important researchers and experts from academia, who are trying to make the bridge with the industry, as always envisioned in the reuse area since McIlroy's pioneer work.

Examples of companies created from academic researchers include *Bayfront Technologies Inc*, a company located in California, founded in 1992 by James Neighbors, one of the pioneers in domain analysis. *Semantic Designs, Inc*, located in Austin, Texas, USA, was founded in 1995 by Dr. Ira Baxter and Dr. Christopher Pidgeon. *BigLever Software, Inc* is also located in Austin, Texas, being founded in 1999 by Charles Krueger, an important expert in software reuse. Other important researchers also have their companies, such as Bill Frakes, with *Software Engineering Guild*, Ruben Prieto-Diaz, with *Reuse, Inc*, Ted Biggerstaff, with *www.softwaregenerators.com*, and Wayne C. Lim, with the *Lombard Hill Group*, all of them working with consulting and services related to software reuse.

Not directly associated to academic researchers, the *Flashline, Inc.* company was a large metadata repository vendor, located in Cleveland, OH, USA. It was purchased in 2006 by *BEA Systems, Inc*, which incorporated Flashline's repository into its product family.

Examples of reuse-dedicated companies from other countries include *The Reuse Company*, located in Madrid, Spain, with tools for knowledge reuse,

INTECS S.p.A., with operational sites in Italy and France, *Pure-Systems*, founded in 2001 in Magdeburg, Germany, being an effort leaded by Danilo Beuche, and two institutions: Otto-von-Guericke-Universitat Magdeburg and the Fraunhofer Instituts Rechnerarchitektur und Softwaretechnik.

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 organizations. 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, organizations 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).

i. 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), (Frakes & Terry, 1996). Poulin's book (1997) does an excellent analysis on software reuse metrics and, in (Mascena et al., 2005), this aspect is revisited in a practical and comparative way;
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 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 a dozen times in order to recover development costs (Favaro, 1991). However, some years after this a more complete study showed that the development costs are recovered after two reuses (Poulin, 1997);

5. **Design for reuse**. Designing for systematic reuse requires processes to be systematic, and domain engineering is a sheer necessity that involves identifying, constructing, cataloging and disseminating a set of software components that can possibly be used in existing or future software, in a particular application domain (Frakes & Isoda, 1994); and
6. **Libraries**. Since 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 library science and Artificial Intelligence (AI), 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.

ii. 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*, *measurement*, and *management*, also highlighted by Frakes & Isoda (see Section above).

iii. 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 a topic. His point of view is that software reuse has good potential, but this has not been achieved yet. While other researchers point out 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 are 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 understand the software reuse scenario one must also consider conceptual reuse thus, architecture rules, frameworks, and patterns show that not only

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, some of the main studies available in the literature are presented.

i. 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, since there are few 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 often 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.

ii. Morisio et al. research (2002), originally the most detailed practical study into software reuse available in the literature, although 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 organizations: 24 such projects conducted from 1994 to 1997 were analyzed using structured interviews.

The projects were undertaken in both large and small organizations, 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*. They 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*.

iii. Rothenberger et al. (2003) 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 source code. 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.

iv. In (Selby 2005), Selby presents a 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 were analyzed. The environment had 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.

v. In (Sherif et al., 2006), based on the argument that even with methods, processes and models to introduce reuse, very few organizations are able to do it, is presented an economic model of systematic reuse implementation within organizations is presented. The model does not present any special way of doing it using, for example, the same aspects previously

discussed such as education, training, incentives, management, processes, and specific tools. At the end, in order to validate the research model, four case studies of software reuse programs were conducted in three multinational US-based organizations. Two of the cases were performed in two software consulting firms, and two in one oil and gas company. Additionally, twenty-six interviews were conducted at all four sites, which were taped and later transcribed. The questions aimed to understand the relationship between successful adoption of systematic reuse, resources, efforts, and yields.

According to (Sherif et al., 2006), the case studies performed provided support to three important contentions: i. for the organization to motivate the divisions to exert additional efforts, *additional resources have to be allocated*; ii. *allocation of additional resources will depend on the situation existing in the divisions*; and iii. *an increase in divisional efforts exerted towards reuse does result in an increase in yields.*

2.11.3. Big Problems in Software Reuse

In this Chapter, the benefits and potentials involving software reuse were presented. However, even when it is achieved, reusing software is not a heaven, when you miss certain aspects of the problem (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 needed.

The software involved is part of the Inertial Reference System (IRS). 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 IRS 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 can be 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 a 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 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			
Selby, 2005			x																	
Sherif et al, 2006							x	x		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 (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.

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), from 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 scientific 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 components, 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 others Practitioners, 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 difficult to achieve, but one way to make it to work is to develop components that can be reused within a domain. Thus with 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.

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

Questions	Answers	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 influence reuse?	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

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 it was 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 unrelated families of systems.

Glass considers that this idea has problem also, because of the complexity in maintaining existing software, the difficulty of understanding the existing solution (even for the developer who originally built the solution) and the hard task in maintaining 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 practitioners have always used is to solve their problems remembering yesterday'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 (Gamma et al. 1995), a description of a recurrent problem, accompanied by a design solution to that problem.

In (Desouza et al., 2006), 25 software development organizations are analyzed to highlight, according to them, the two missing parts of the reuse knowledge management (KM) puzzle – recoding and redesign. Thus, during the course of two years about 100 software engineers in over 25 organizations were interviewed (semi-structured interviews with approximately 45 minutes). In five organizations, onsite observations of how software engineers worked with code modules were also conducted. Moreover, three organizations allowed their software engineers to participate in a survey that gauged antecedents to reuse of knowledge objects. Of the 25 organizations, 12 were ranked in the Fortune 100 list, and the rest were medium-size IT firms.

In general, the findings show that rookies are more likely to reuse software artifacts while experts refrain from doing so. More reuse occurs within than across software development groups and transient teams are more likely to engage in reuse than permanent development teams.

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, some of the most relevant and active software reuse efforts around the world will be presented.

2.14.1. Software Reuse in North America

i. From 1988 up to 1996, in United States, Mark Simos with a number of colleagues from organizations including Lockheed Martin, Hewlett Packard, the DARPA STARS Program, Change Management Consulting, and Synquiry

Technologies, Ltd. developed a corpus of methodology materials 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)[4] project produced several documents on methodology for software reuse. Many of these 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 do not contain any information available on the Web site, only scientific reports relating to the usage of this project cited above.

ii. From 1994 to 2000, the Advanced Technology Program (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) in United States.

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 work, two are even 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.

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

2.14.2. Software Reuse in South America

During the years 2004-2006, the Brazilian government invested about three million dollars in reuse projects involving the definition of methods to develop

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

and certify reusable assets, the development of tools, web services and components in strategic domains, besides a public component repository system and a business model for it[5]. These projects are composed of important universities and organizations in the country.

Initial results of these projects are promising, as the development of courses (disciplines and commercial courses), dissertations and thesis, specialized human resources, workshops, papers, and reuse products with high quality. The Brazilian government and researchers believe that university and industry working together can be the way of improving the reuse practices and impacts in industrial context and hoping seeing it soon.

2.14.3. Software Reuse in Europe

i. The Engineering Software Architectures, Processes and Platforms for Systems-Families (ESAPS)[6] project started on 1999, and 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 organizations, research and technology transfer institutions and universities.

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

Until today, many workshops (these workshops will create the International Workshop on Product Family Engineering (PFE), currently called International Conference on Software Product Lines (SPLC)) and definitions of processes, methods and models for software reuse were developed. Some real

[5] http://www.mct.gov.br/upd_blob/0001/1080.pdf
[6] http://www.esi.es/en/Projects/esaps/esaps.html

case studies of these proposals were developed in several European organizations, 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.

ii. From Concepts to Application in System-Family Engineering (CAFÉ) started in 2001 and aims to provide a path from ESAPS concepts to application. CAFÉ is carried out by a consortium of leading European organizations, 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 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 organizations 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[7].

iii. According to 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.

McGibbon et al., based on their 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

[7] http://www.esi.es/en/Projects/Cafe/cafe.html

managed as well as 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, the organizations eventually found that they needed to focus at the project level to explore and understand the issues involved with CBSE.

Another 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 system 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.

2.14.4. Software Reuse in Asia

i. 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) in the area. After an investigation and planning phase, 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 organizations 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 organizations 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. This year, APSEC will be focused on CBD.

ii. 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[8]. However, this initiative is extensively cited in some technical papers and computer-science workshops, thus, probably, the impact of this initiative on the emergence and competitiveness of the Korea software industry can be important in the future.

iii. According to Aoyama et al. (Aoyama et al., 2001), around 1995, a set of research and development programs on CBSE were initiated in Japan. These initiatives can be summarized as follows:

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

- From 1996 to 1998, 21 major computer and software organizations, 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. Currently, this workshop is the main conference about CBSE around the world;
- From 1998 to 1999 the Information Technology Consortium (ITC) 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, Japan Information Service Industry Association (JIISA), the largest industry association of software development organizations in Japan, formed a working group to transfer CBSE technology to its 600-odd member-organizations;
- For a long time, as shown early, the Japanese software industry has been trying to adopt 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[9].

As shown, CBSE is a strategic research and development issue in Japan and Aoyama believes that CBSE will provide the competitive edge that will enable software development organizations to more rapidly produce quality software.

[9] Currently, Vector Inc. website is not available in the Internet.

2.15. Key Developments in the Field of Software Reuse: Summary

Figure 2.1 summarizes the timeline of research on the software reuse area[10], in which are marked (with an "X") the milestones of research in the area. Initially, 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 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 strongly influenced the research in the component retrieval area.

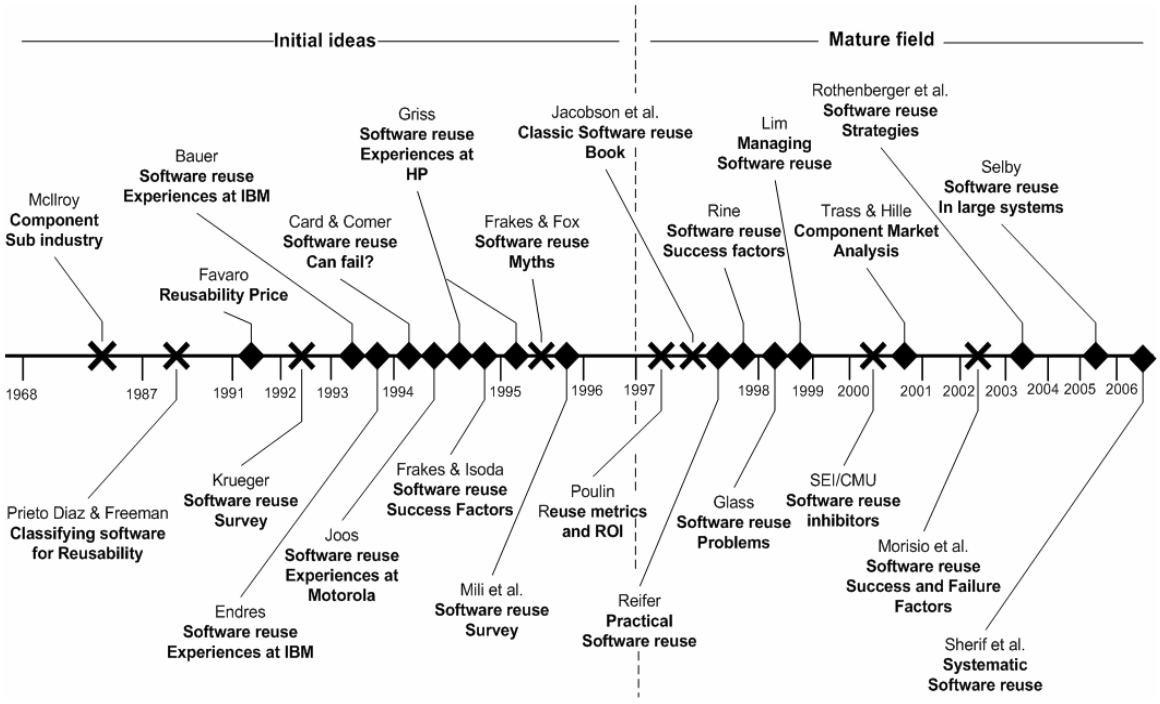

Figure 2.1. Research on Software Reuse Timeline.

In 1992, the first important survey on software reuse was presented. Krueger (1992) presented the eight different approaches to software reuse found

[10] 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.

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 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 (1997), Jacobson et al. (1997), Reifer (1997), and Lim (1998) among others. 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. 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. 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 to the software reuse success. However, some practices need more research and findings to prove the success, such as formal processes, 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 organizations 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 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 presented several issues related to software reuse adoption on larger-scale. Among the considerations discussed the following points were highlighted*:* i. *education in software reuse area still is a weak point;* ii. *the necessity of the academia and industry to 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 often 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 presented a summary on the software reuse research discussing unsolved problems, based on the Eighth International Conference on Software Reuse (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 organizations interested in improving software development quality and productivity, and in costs reduction. In this Chapter, a survey on software reuse was presented, discussing

its origins, company reports, success and failure factors, myths, and inhibitors for software reuse, in order to guide organizations towards systematic software reuse.

The Chapter also identified essential aspects in the software reuse program (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.

The next Chapter presents the state-of-the-art on the software reuse processes area discussing their origins, fundamentals, results, and strong and weak points in order to define a base for the approach defined in this work.

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

"Those who cannot remember the past are condemned to repeat it"

George Santayana (1863 – 1952)
Spanish Philosopher, essayist, poet, and novelist

During the last years, software has conquered an essential and critical role in our society, creating dependencies on the features and services offered through computerized systems. Currently, any modern product or service embeds and/or exploits some piece of software (Fuggetta, 2000).

Unfortunately, software applications are complex products that are difficult to develop and test and, often, software presents unexpected and undesired behaviors that may even cause severe problems and damage (Leveson & Turner, 1993). For these reasons, researchers and practitioners have been paying increasing attention to understanding and improving the quality of the software being developed. It is accomplished through a number of approaches, techniques and tools (Boehm, 2006) and one of the main directions investigated is centered on the study and improvement of the process through which software is developed (Osterweil, 1987).

There are several definitions on software process (Osterweil, 1987), (Pressman, 2005). According to Sommerville (2006), a *software process is a set of activities that leads to the production of a software product*. Ezran et al. (2002) consider that 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 the tasks necessary to produce and manage software, whereas **reuse processes** are the subset of tasks necessary to develop and reuse assets (Ezran et al., 2002, pp. 65).

The adoption of either a new, well-defined, managed software process or a customized one is a possible facilitator for success in reuse programs (Morisio et al., 2002). However, the choice of a specific software reuse process is not a trivial task, because there are technical (management, measurements, tools, etc) and non-technical (education, culture, organizational aspects, etc) aspects that must be considered.

This Chapter presents fifteen software reuse processes representing the state-of-the-art of the area (Almeida et al., 2005a) and discusses the requirements and important issues that systematic reuse processes must consider. It is important to highlight that although the reuse literature does not define a common taxonomy for approaches, processes, methods, and methodologies as will be seen next, this thesis considers all of them as reuse processes, once that they agree with the definition presented in (Ezran et al., 2002) for it.

3.1. Introduction

As discussed in the previous Chapter, over the years, several research works, including company reports (Bauer, 1993), (Endres, 1993), (Griss, 1994), (Joos, 1994), (Griss, 1995), 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* (Domain Engineering)

and *with* (Application Engineering) reuse, and lack 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 roles) 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).

With this motivation, the reminder of this Chapter presents a detailed review based on fifteen software reuse processes representing the state-of-the-art of the area in order to understand the crucial points in software reuse processes.

3.2. 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 (education, culture, organizational aspects, etc), must describe two essential activities: the development *for* reuse (Domain Engineering) which will be discussed next and the development *with* reuse (Application Engineering) which consists in building applications based on the assets produced in Domain Engineering.

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, **Software Product Lines**, as can be seen in the next Sections.

3.2.1. Domain Engineering Processes

Domain Engineering (DE) *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, pp. 20).

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), (Villela, 2000) a special effort is put into the domain engineering processes in order to develop reusable software.

3.2.1.1. 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* (DSL), 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 due to the complexity of performing activities such as writing transformations and using the Draco tool. Even with some advances related to his work (Leite, 1994), many of these problems still remain unsolved.

3.2.1.2. Conceptual Framework for Reuse Processes (CFRP)

In this context (how to perform Domain Engineering in a systematic way), 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 *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 Application Generators*.

3.2.1.3. 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.

Before Simos' work, the research involving software reuse processes was too general and did not 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.

3.2.1.4. Reuse-driven Software Engineering Business (RSEB) and FeatuRSEB

With this motivation, 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 the UML notation. The method was designed to facilitate both the development of reusable object-oriented software and software reuse. Similar to the Unified Process (UP) (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 modeling. Moreover, the process does not describe a systematic way to perform the asset development as proposed. Another shortcoming of RSEB is the lack of feature models to perform domain modeling considered a key aspect by the reuse community (Kang et al., 1990). 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 Feature-Oriented Domain Analysis (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 RSEB were not solved such as the lack of a systematic way to achieve the asset development.

3.2.1.5. Feature-Oriented Reuse Method (FORM)

Concurrently with Griss et al.'s work, an important 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 (DSSA), 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*.

FORM does not discuss 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 identification, specification, design, implementation and packaging of the components are explored a little.

After the domain engineering, the application engineering process is performed. Once again, the emphasis is on the 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.

3.2.1.6. Odyssey-DE

Based on the analysis of the domain engineering approaches and the application development necessities at that time, Villela (2000) concluded that the none of the approaches are not completely closed to reuse necessity. In this context, she presents the Odyssey-Domain Engineering (DE) process, focused on the integration between the reuse aspects and domain understanding from DE's approaches and the systematic sequence of activities defined in the component-based development processes.

Odyssey-DE is composed of five steps: *Planning*, *Risk Analysis*, *Domain Analysis*, *Domain Design*, and *Domain Implementation*, however, in her thesis, just the last three steps are explored. The key aspect of Odyssey-DE is its integration with the Odyssey environment and all reuse infra-structure available. Although, the process presents lack of details in some steps such as domain analysis, where the activities of domain scoping and validation are little explored, and in domain design, where the activities for component identification and grouping are not discussed.

3.2.2. 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). According to (Clements & Northrop, 2001, pp.05), a *software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way*. 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 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.

3.2.2.1. Product Line Software Engineering (PuLSE)

Based on the mentioned limitations of domain engineering and lack of product-lines approaches, Bayer et al. (1999) 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. For this task, PuLSE has components for Customization (BC), Scoping (Eco), Modeling (CDA), Architecting (DSSA), Instantiating (I) and Evolution and Management (EM). 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 does not require a specific design methodology or a specific *Architecture Description Language* (ADL). We do not agree with this vision because the 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.

3.2.2.2. Family-Oriented Abstraction, Specification and Translation (FAST)

Based on industrial experiences in software development, especially at Lucent Technologies, David Weiss & Chi Lai presented the Family-Oriented Abstraction, Specification and Translation (FAST) process (Weiss & Lai, 1999). Their goal is to provide a systematic approach to analyze potential families and to develop facilities and processes for generating family members. FAST defines a pattern for software production processes that strives to resolve the tension between rapid production and careful engineering. A primary characteristic of the pattern is that all FAST processes are organized into three sub-processes: *Domain Qualification* (DQ), *Domain Engineering*, and *Application Engineering*.

Domain Qualification consists of an economic analysis of the family and requires estimating the number and value of family members and the cost to produce them. *Domain Engineering* makes it possible to generate members of a family and is primarily an investment process; it represents a capital investment in both an environment and the processes for rapidly and easily producing family members using the environment. *Application Engineering* uses the environment and processes to generate family members in response to customer requirements (Weiss & Lai, 1999).

The key aspects of FAST is that the process is derived from practical experiences in industrial environments, the systematic definition of inputs, outputs, steps, roles, and the utilization of a process model that describes the process. However, some activities in the process such as in *Domain Engineering* are not as simple to perform, for example, the specification of an Application Modeling Language (AML) – language for modeling a member of a domain – or to design the compiler to generate the family members.

3.2.2.3. Komponentenbasierte Anwendungsentwicklung (KobrA) Approach

As described in Section 3.2.2.1., in circumstances where there were no pre-existing processes or well-defined products, the introduction of PuLSE turned

out to be problematic. In 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 systematic 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.

3.2.2.4. 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 (Ommering, 2005). 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 why 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 sub-processes: *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 sub-processes 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 sub-processes. 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.

3.2.2.5. SEI's Framework for Software Product Lines

According to (Czarnecki & Eisenecker, 2000), in order to successfully introduce and run domain and application engineering in an organization, it is necessary

to address a wealth of issues that go beyond the scope of current DE methods. In particular, they highlight that there are management and organizational issues concerning establishing both processes and the feedback loop between them. Additionally, there are questions about how to build a business case for a transition to a reuse approach, how to launch and institutionalize it, manage risks, etc, as explained in Chapter 2. Moreover, other points should be addressed such as market analysis and technology forecasting, support for deciding between development, purchase, mining, and outsourcing of assets, besides specific tools and processes.

These different issues, discussed briefly in the previous paragraph, are part of the *Framework for Product Line Practice*, proposed by the SEI. According to (Clements & Northrop, 2001), there are three essential product line activities: *core asset development*, *product development* and *management*. The *core asset development* goal is to establish a production capability for products. *Product development* consists in developing the products reusing the assets. At the end, *Management* – technical and organizational – must be strongly committed to the software product line's success.

The framework defined by SEI is very interesting and robust, involving important technical and non-technical aspects grouped in software product line practical areas; however, the framework is very generic and does not define specific ways of performing the activities. Their approaches consist in presenting the practical areas such as architecture definition or evaluation, for instance, and discussing the specific risks and practices compared to single software development.

3.2.2.6. 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 difficult 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 achieved 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 constructed in an isolated way, without this view.

3.2.2.7. Form's extension

In (Kang et al., 2002), 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.

3.2.2.8. Product Line UML-Based Software Engineering (PLUS)

The most current process related to product lines is seen in (Gomaa, 2005). PLUS extends the UML-based methods for single systems development to support software product lines providing modeling techniques and notations for product line engineering. According to Gomaa, the method is an iterative object-oriented software process that is compatible with the Unified Software development process (Jacobson et al., 1999) and the spiral process model (Boehm, 1988).

In PLUS, initially, three general steps are performed: *Requirements*, *Analysis*, and *Design Modeling*. First, for the requirements engineering activity, use case modeling and featuring modeling are provided. Next, for the analysis activity, static modeling, dynamic interaction modeling, dynamic state machine modeling, and feature/class dependency modeling are introduced. Finally, for the design activity, software architecture patterns and component-based software design are proposed. Once concluded these steps, application engineering is performed.

PLUS is an interesting method to software product lines based on UML with a low learning curve and systematic steps to perform it, especially, in requirements and analysis steps. However, in *Requirements* and *Analysis*, activities related to scoping are not considered, and, in *Design*, PLUS does not define how to identify and group components of the architecture. Moreover, issues related to component implementation and documentation are not discussed.

3.3. Towards an Effective Software Reuse Process

Figure 3.1 summarizes the timeline of research on the software reuse processes area, in which the milestones of research in the area are marked (with an "X"). 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 the STARS project.

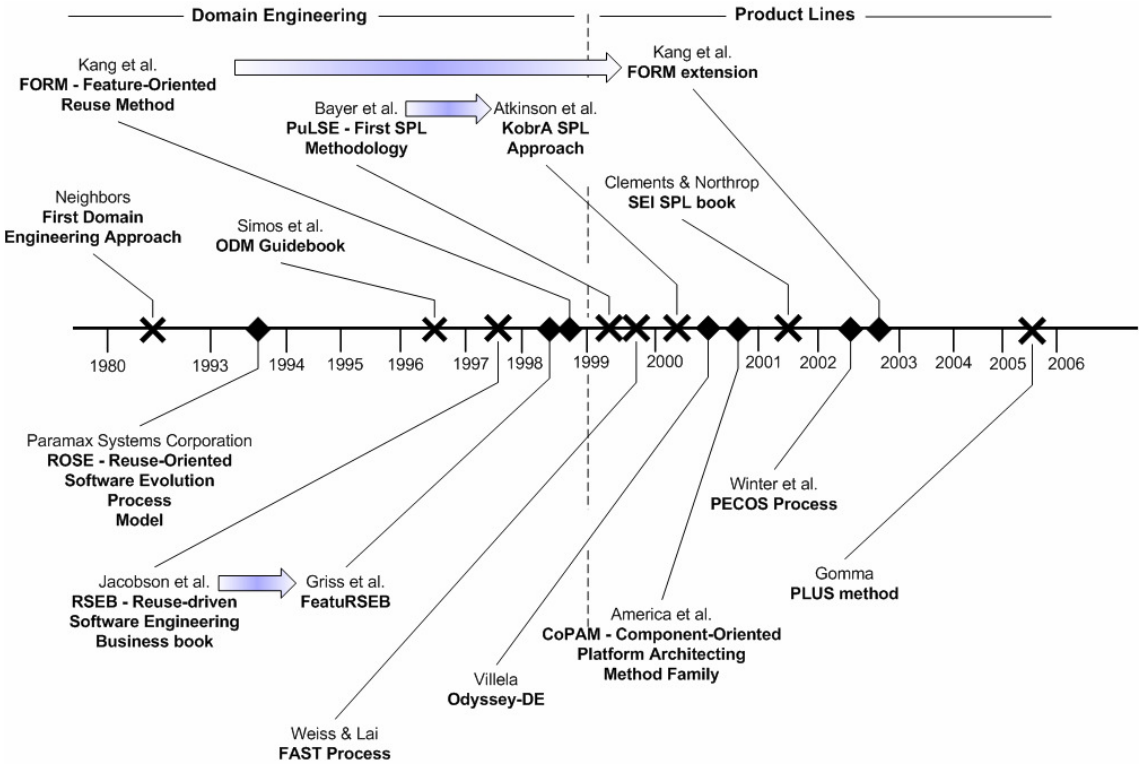

Figure 3.1. Research on software reuse processes timeline.

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) and after explored also in (Weiss & Lai, 1999), (Atkinson et al., 2000), (Gomaa, 2005).

Currently, 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 systematically perform 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, these questions are analyzed by presenting a set of requirements for an effective software reuse process. The approach to define the requirements was composed of three steps: initially, the one was defined based on the state-of-the-art of the reuse processes; next, these requirements were discussed with other researchers from the RiSE group; and, finally, they were refined according to RiSE's experience in industrial reuse projects. It is not a definitive set, nevertheless we believe that the identified requirements constitute a solid basis for a reuse process.

3.3.1. Domain Engineering

As discussed in details in Chapter 2 and presented also in this Chapter, Domain Engineering is a key aspect for software reuse. However, the reuse processes based on DE or SPL present gaps among the steps of analysis, design, and implementation. According to 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.

3.3.2. Application Engineering

The development *with* reuse 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.

3.3.3. Metrics

Since the time that the concept of software reuse was initially 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 (Poulin, 1997).

In this way, some fundamental questions in this context are then: “how much software reuse?” and “how much better with software reuse?” (Mascena et al., 2005). These questions lead to another important software engineering area closely related to software reuse: software 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. Thus, a reuse process must define what, where, and when to measure. However, metrics are often neglected in existing reuse processes.

3.3.4. Economics

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.

3.3.5. Reengineering

Software reengineering is being used to recover legacy systems and allow their evolution (Jacobson & 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 design, code and documentation. However, even being an important issue, the current reuse processes do not present aspects related to how to reengineer systems in order to be reused.

3.3.6. Adaptation

Many organizations are actively pursuing software process maturity models in order to increase their competitiveness. Usually, they use measures such as the Capability Maturity Model (CMM) to structure their process improvement initiatives. On the other hand, one method often cited for accelerating process improvement within an organization is to replicate a standard, reusable organizational process within other projects. The work reported in (Hollenbach & Frakes, 1996) defined this approach as a reusable process. According to them, 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. However, aspects for reuse process adoption are neglected in reuse processes, even with important empirical research (Morisio et al., 2002) showing the importance of modifying non-reuse process for successful in reuse projects.

3.3.7. Quality

As widely discussed in the previous Chapter, an important issue in software reuse is the quality of the reusable assets. However, to fulfill this goal, it is necessary to have reliable reusable assets. This reliability can be certified, for example, using testing by developing usage models, applying usage profiles, formal methods or applying a certification model to show the level of confidence in them (Meyer et al., 1998).

Thus, quality is an important aspect of reuse processes. However, the available reuse processes do not discuss, for example, ways to test, inspect and certify the produced assets.

3.4. Summary of the Study

Table 3.1 shows the relation between the works described in Section 3.2 and the requirements from Section 3.3. Each number corresponds to a work: **1. Neighbors (Draco)**, **2. ROSE PM**, **3. ODM**, **4. RSEB**, **5. FeatuRSEB**, **6. FORM**, **7. Odyssey-DE**, **8. PuLSE**, **9. FAST**, **10. KobrA**, **11. CoPAM**, **12. SEI**, **13**. **PECOS**, **14. FORM's extension**, **15. PLUS**. In the table, a white circle indicates that the requirement is partially satisfied by the work. Gaps show that the requirement is not even addressed by the work.

Table 3.1. Relation between the works on software reuse processes and the requirements.

Requirements	Works														
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Domain Engineering	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
Application Engineering	○	○		○	○	○	○	○	○	○	○	○	○	○	○
Metrics															
Economics															
Reengineering															
Adaptation		○	○	○				○		○		○			
Quality															

Analyzing Table 3.1, it can be seen that there are gaps in proposed software reuse processes and important requirements are not considered. All

the requirements are important and relevant for the success of a reuse process; however, in this thesis the focus is on Domain Engineering, which is considered as a key aspect to success in reuse, especially when considering software reuse processes.

3.5. Chapter Summary

Adopting a software reuse process can be 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, fifteen software reuse processes corresponding to the state-of-the-art of the area were discussed. 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. Additionally, based on the weaknesses and strengths of existing processes, a set of requirements for an effective software reuse process was presented. In this context, the next Chapter presents the Domain Engineering process proposed by this work.

RiDE: The RiSE Process for Domain Engineering

"...reuse is neither a silver bullet nor a magic weight loss pill. It is a diet and exercise program..."

Paul G. Bassett (1997)
Software Reuse Specialist

Currently, the trend in software development is toward bigger and more complex systems (Boehm, 2006), (Broy, 2006). This is due in part to the fact that computers become more powerful every year, leading users to expect more from them. Moreover, this trend has also been influenced by the expanding use of the Internet for exchanging all kinds of information (texts, pictures, multimedia, etc). Thus, some characteristics are strongly wished by managers, software engineers, and users, such as software that is better adapted to their needs, with short time-to-market and low cost (Jacobson et al., 1999). However, this demand for more powerful and complex software has not been matched with how software is developed.

According to Jacobson et al. (1999), most people develop software using the same methods that have been used for the last twenty-five years. Thus, unless they update these methods, the software development community will probably not be able to accomplish the goals of developing the complex software needed.

In this context, the software development community needs a controlled and disciplined way of working, as well as a process that integrates the many facets of software development. It needs a common approach, a process that:

- provides guidance to the order of a team's activities;
- directs the tasks of individual developers and the team as a whole;
- specifies what artifacts should be developed; and
- offers criteria for monitoring and measuring project products and activities.

As widely discussed in the previous Chapters, software reuse can be an important way to develop complex software, offering benefits related to quality, productivity and costs, mainly, using a well-defined reuse process. However, the current software reuse processes present gaps and lack of details in key activities such as, for example, domain engineering. Other requirements like application engineering, metrics, economic aspects, reengineering, adaptation, and quality are important as well, but the focus on this thesis is on domain engineering.

In this context, this Chapter presents an overview of the proposed domain engineering process, its foundations, elements and steps.

4.1. Introduction

According to Weiss & Lai (1999), software development staff faces a continuing dilemma. On one hand, they are asked to create software that attracts customers with its functionality, ease of use, and reliability, and that is easy to enhance and evolve as customer needs and technology changes. On the other hand, they are pressured to produce software so that it can be marketed ahead of the competition, and, often, the pressure for rapid production is often the enemy of careful engineering. By failing to pay attention to activities that promote faster development of other versions of the same product, or of similar products, the same efforts for each new version or similar product are repeated.

Thus, attaining resolution between schedule pressure and careful engineering is rarely a consideration in current software development

processes. However, changing this analysis for other areas the same pressures can be found. In fields such as aerospace engineering (Boeing), automotive engineering (Henry Ford), and computer engineering (HP), methods of rapidly producing carefully engineered products have long been explored and continue to improve (McGregor et al., 2002). Many of these methods are based on the idea of developing a family of products that can be produced using production facilities.

In many cases, each member of a family is a complete product, such as a family of automobiles in which each automobile is a member of the family and is a complete product in itself. In other cases, each item is part of a product and is intended to fit into a larger family, such as a tire for an automobile. As explained in the previous Chapter, a family of products designed to take advantage of the common aspects and predicted variabilities is called a product line. Although software development is quite different from development of airplanes, automobiles, or computers, software engineers and others can benefit significantly from applying family-based production strategies. Moreover, according to Weiss & Lai (1999), most software development is mostly redevelopment.

In the software world, this idea was firstly introduced by David Parnas (1976) when he said (pp. 01): "*We consider a set of programs to constitute a family, whenever it is worthwhile to study programs from the set by first studying the common properties of the set and then determining the special properties of the individual family members*".

The first step in the transition – from single systems to system families – is to adopt a domain engineering or software product line process (Czarnecki & Eisenecker, 2000), once they realize software engineering based on system families and can help to answer questions such as: "What is the right size for a system family?" and "How do you analyze and represent the commonalities and variabilities among the members of a system family?".

The software reuse community does not always have a clear distinction between domain engineering and software product lines. Some authors consider domain engineering as part of the software product line process, corresponding to core asset development (Clements & Northrop, 2001). On the other hand, for

researchers such as Bill Frakes (Frakes & Kang, 2005) it is the same idea with new denominations. The goal of this thesis is not to discuss these different points of view. Instead, it focuses on domain engineering, a consensus between researchers from the reuse and the software product line communities.

4.2. Overview of the Process

As defined in the previous Chapter, 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 (i.e. reusable work products), as well as providing an adequate means for reusing these assets (i.e. retrieval, qualification, dissemination, adaptation, assembly, and so on) when building new systems (Czarnecki & Eisenecker, 2000, pp. 20).

A domain engineering process should define three important steps: **Domain Analysis (DA)**, **Domain Design (DD)**, and **Domain Implementation (DI)**. In general, the main goal of Domain Analysis is domain scoping and defining a set of reusable, configurable requirements for the systems in the domain. Next, Domain Design develops a common architecture for the system in the domain and devises a product plan. Finally, Domain Implementation implements the reusable assets, for example, reusable components, domain-specific languages, generators, and a production process (Czarnecki & Eisenecker, 2000).

Before presenting more details about the proposed domain engineering process, it is important to discuss some basic concepts which are used further:

- **Domain:** The term domain is assigned with different meanings in different disciplines and communities, such as linguistics, cultural research, Artificial Intelligence (AI), object-oriented programming, and software reuse (Czarnecki & Eisenecker, 2000). Mark Simos et al. (1996) distinguish two general categories of this term: **i.** Domain as the real world. **ii.** Domain as a set of systems. A domain as the real world encapsulates the knowledge about a problem area (e.g., a starship game domain includes concepts such as starships, projectiles, etc), but not about the software automating and supporting the processes in the problem area,

which is a separate entity. This notion of a domain is also used in OO and AI communities. In the software reuse community, the term domain encompasses not only the real world knowledge in a given problem area, but also the knowledge about how to build software systems in that area, corresponding to the domain as a set of systems view. In RiDE, the second definition will be used, since it is the consensus in the reuse community. During the following Chapters, which will present the process, the game domain will be used to show its steps, inputs and outputs.

- **Feature:** Features and Feature Models are widely used in domain analysis to capture the commonalities and variabilities of systems in a domain (Kang et al., 1990). In general, there are two definitions of feature found in the reuse literature: **i.** An end-user visible characteristic of a system, the FODA definition. **ii.** A distinguishable characteristic of a concept (e.g. system, component, etc) that is relevant to some stakeholder of the concept, the ODM definition (Simos et al., 1996). In RiDE, the first definition will be used, since it is the basis for the domain analysis area.

- **Applications:** In the reuse community, some practitioners use a different set of terms to convey essentially the same ideas (Northrop, 2002). Some refer to a product line as a product family (America et al., 2000), to the core asset as a platform and domain engineering (Pohl et al., 2005) and products as applications. In RiDE, products will be considered as applications. Core asset and product development will be referred, respectively, as domain and application engineering, as widely used by the reuse community, especially in the context of domain engineering.

Other important concepts, more specific to each step of the process, are presented later, together with the process detailed description.

4.3. The Foundations

A software development process can be understood as the set of activities needed to transform user requirements into a software system. The way it is done can change from process to process. For example, processes can be focused on domain engineering, services (Papazoglou & Georgakopoulos, 2003), or Model-Driven Development (MDD) (Schmidt, 2006), or use different process models; however, all kinds of processes are based on some foundations. In this Section, the foundations for the domain engineering process will be presented.

i. Process Model. A software process model is an abstract representation of a software process (Sommerville, 2006). Each process model represents a process from a particular perspective, and thus provides only partial information about that process. There are different process models published in the software engineering literature, such as Waterfall model, Evolutionary development, Component-Based Software Engineering (CBSE), Incremental delivery, Spiral development, among others (Pressman, 2005). These process models are widely used in current software development practice. Moreover, they are not mutually exclusive and are often used together, especially for large systems development. The Rational Unified Process (RUP), for example, combines elements of all of these models.

RiDE is based on the spiral process model (Boehm, 1988), however, it presents some characteristics of the CBSE process model, since reusable assets are used to develop applications. The spiral model proposed originally by Boehm (1988), rather than representing the software process as a sequence of activities with some backtracking from one activity to another, consists of a spiral, where each loop represents a phase of the software development process. Each loop in the spiral is split into four sectors: **Objective setting** (specific objectives for that phase of the project are defined, constraints are identified and a management plan is defined), **Risk assessment** and **reduction** (for each risk, a detailed analysis is performed with steps to reduce it), **Development** and **Validation** (after risk evaluation, a development model for the system is chosen), and, finally, **Planning** (the project is reviewed and a decision about whether to continue with a further loop of the spiral or not is made). In the domain engineering process, issues for planning and risk

assessment are out of the scope of this thesis. However, the steps for domain analysis, domain design, and domain implementation complete the spiral. Figure 4.1 shows an overview of the process according to the spiral process model.

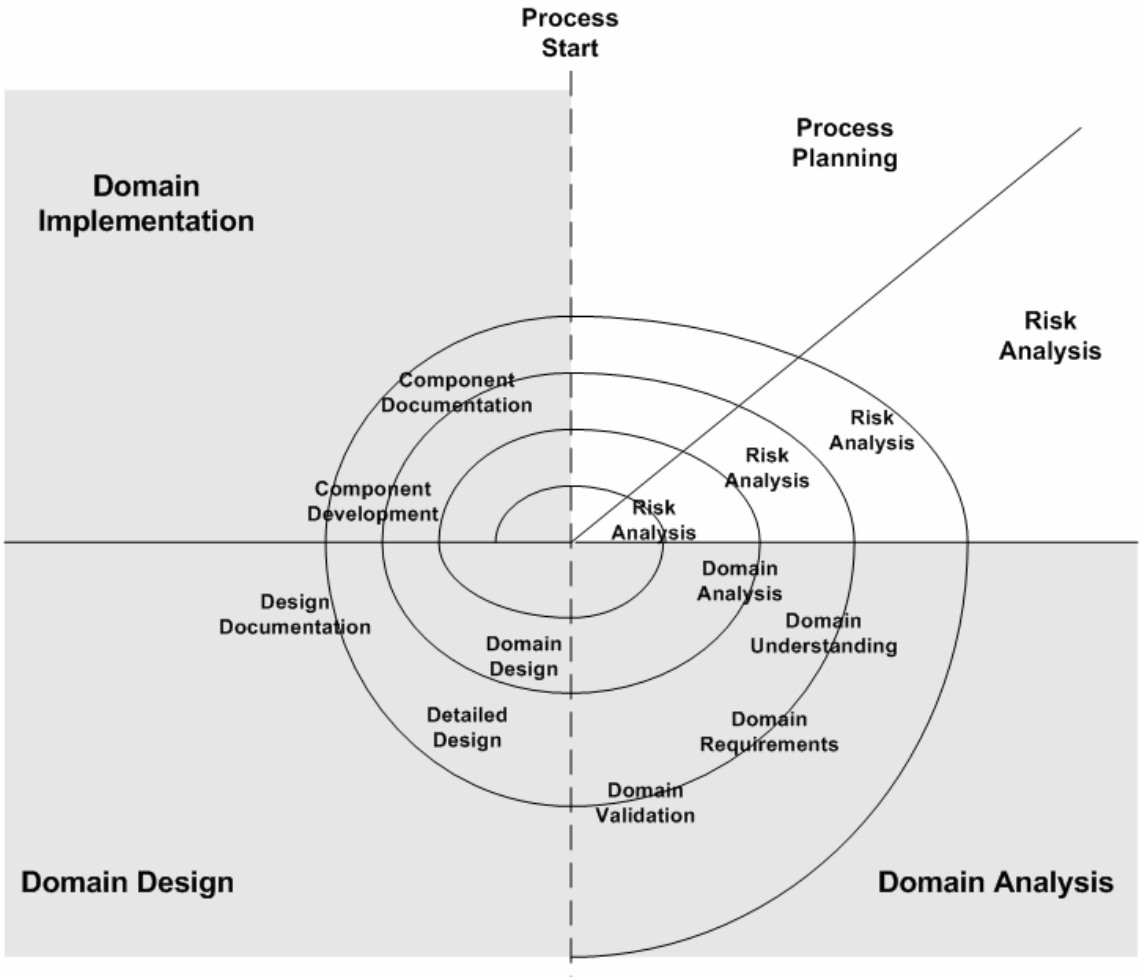

Figure 4.1. Process Model of the Domain Engineering Process.

ii. Domain Driven. Instead of traditional software development processes as, for example, the RUP, which is use-case driven, the domain engineering process is domain-driven, where the focus is on a set of applications for a particular domain. In this domain, the crucial points are: to identify common and specific features from existing, future, and potential applications; to organize this information in a domain model; next, to design the Domain-Specific Software Architecture (DSSA); and, finally, to implement reusable components for that domain. Even being domain driven, use cases are also used in the process as will be shown in Chapter 6.

iii. Iterative and Incremental. Developing a commercial software product is a large undertaking that may continue over several months to possibly a year or more. It is practical to divide the work into smaller slices or mini-projects where each mini-project is an iteration that results in an increment (Jacobson et al., 1999). Iterations refer to steps in the workflow, and increments, to growth in the product. To be effective, the iterations must be controlled; in order to be selected and carried out in a planned way.

In reuse projects, this issue is even more critical, since a software product line, for example, does not stop its development until it is obsolete and is not used for new products anymore (Svahnberg et al., 2001). Thus, RiDE should also be iterative and incremental.

iv. Software Architecture. Software architecture involves the structure and organization by which modern system components and subsystems interact to form systems; and the properties of systems that can be better designed and analyzed at system level (Kruchten et al., 2006). Additionally, a software architecture separates the overall structure of the system, in terms of components and their interconnections, from the internal details of the individual components (Shaw & Garlan, 1996). In the context of software reuse processes, software architectures play a role that is even more important, since the goal of domain design is to design a software architecture which supports several applications in a domain. Thus, concepts present in software architectures such as modules, information hiding, parameterization, and abstraction should be strongly used in the domain engineering process.

v. Component-Based Development (CBD). In the early 1990's, it became apparent to both researchers and practitioners that object-oriented technologies were not enough to cope with the rapidly changing requirements of real-world software systems. One of the reasons was that, although object-oriented methodologies encourage one to develop rich models that reflect the objects of the problem domain, this does not necessarily yield software architectures that could be easily adapted to changing requirements. In particular, object-oriented methods does not typically lead to designs that make a clear separation between computational and compositional aspects (Schneider & Han, 2004). Even dating back to McIlroy in 1968, component-based

development emerged in the 1990's, motivated by designer's frustration on the discussed issues and by the fact that object-oriented development had not led to extensive reuse, as originally suggested. This made marketing objects as reusable assets difficult and no significant market for individual objects has ever developed (Traas & van Hillegersberg, 2000), (Szyperski, 2002). In the context of software reuse processes, component-based development techniques are important because in domain design, for example, it is interesting to modularize the architecture in well-defined components, which can be easily changed without affecting other parts of the architecture. Moreover, in domain implementation, whose goal is to develop reusable assets, an important way of doing it is through a set of domain-specific components, increasing the reuse potential.

vi. Design Patterns. Design patterns were derived from ideas put forward by Christopher Alexander (Alexander et al., 1977), who suggested that there were certain patterns of building design that were common and that were inherently pleasing and effective. The pattern is a description of the problem and the essence of its solution, so that the solution may be reused in different settings. In the software world, design patterns were popularized by the gang of four (Gamma et al., 1995). According to Gamma and his colleagues, design patterns describe a recurring design problem to be solved, a solution to the problem, and the context in which that solution works. A design pattern is a larger-grained form of reuse than a class because it involves more than one class and the interconnection among objects from different classes. From the perspective of the domain engineering process, design patterns are important because they can be used to encapsulate the variability existing in domain analysis model and perform the mapping for design.

4.4. Elements of the Process

As previously stated, a process is a sequence of decision-making activities. For any engineering process, the people need to know answers for issues such as (Weiss & Lai, 1999):

- Who should do what?
- When do they do it?

- How do they do it?
- What should they produce?
- What resources do they need?
- How do they know when they're finished?

In order to answer such questions and to offer a strong basis for being used, the domain engineering process is composed of several elements, such as:

i. Guidelines. In specific situations, the process should describe systematic ways of performing some activities;

ii. Principles. According to the Oxford dictionary, *a principle is a law, a rule or a theory that something is based on* (Oxford, 2000, pp. 1004). In the domain engineering process, principles can be understood as important concepts and techniques that guide the process.

iii. Language. In order to be better understood and used, a process needs to describe a language that aids users to use it effectively. A good description is also useful to repeat the process for many different software development projects and offer a reference for it. Currently, this issue is very important because as systems become bigger and more complex, the process used to produce them must provide well-defined steps, activities and sub-activities in their production. In RiDE, the Structured Analysis (SA) language is used (Ross, 1977). According to Ross, SA derives from the way our minds work, and from the way we understand real-world situations and problems, structuring language for communicating ideas more effectively. SA's thesis is that the human mind can accommodate any amount of complexity as long as it is presented in easy-to-grasp chunks that are structured together to make the whole.

The main facts about SA are as follows (Ross, 1977):

- It incorporates any other language; its scope is universal and unrestricted;
- It is concerned only with the orderly and well-structured decomposition of the subject matter;

- The decomposed units are sized to suit the modes of thinking and understanding of the intended audience;
- Those units of understanding are expressed in a way that rigorously and precisely represents their interrelation;
- This structured decomposition may be carried out to any required degree of depth, breadth, and scope while still maintaining all of the above properties; and
- SA increases both the quantity and quality of understanding that can be effectively and precisely communicated well beyond the limitations inherently imposed by the imbedded natural or formal language used to address the chosen subject matter.

The actual output of SA is a hierarchically organized structure of separate diagrams, each of which exposes only a limited part of the subject to view, so that even very complex subjects can be understood.

Besides the presented aspects, SA was used successfully in previous domain engineering process such as Draco, ROSE PM, and ODM.

iv. Roles. People fill many different positions in a software development organization. Preparing them for these positions takes education and pin-pointed training followed by careful assignment supported by mentoring and helpful supervision (Jacobson et al., 1999). A role can be used to represent a unit of responsibility, assignment, authority, or work force. Additionally, roles can be used to describe organizations and to give work assignments to people enacting a process, indicating who can perform an activity.

v. Artifacts. Artifacts represent final or intermediate work products or the information needed to produce them, such as documents, models, design specification, code, or any useful representation of information.

vi. Activities. An activity is an operation that is performed on one or more artifacts and produces a useful output. Sometimes, an activity can be divided into sub-activities due to its granularity. Activities indicate what must be done and who can do it.

4.5. Steps of the Domain Engineering Process

The RiDE process for Domain Engineering is composed of three steps**: Domain Analysis**, **Domain Design**, and **Domain Implementation**. Due to the amplitude, each step is respectively divided in activities and sub-activities. The steps are defined to be used in sequence. However, even with less optimal results, they can be used separately. Therefore, in RiDE, each step will be treated as an approach for a different part of the domain engineering life cycle. In this sense, there is an approach for Domain Analysis, described in Chapter 5, an approach for Domain Design, described in Chapter 6, and, finally, an approach for Domain Implementation, described in Chapter 7.

4.6. Chapter Summary

As widely discussed in Chapter 2, one of the approaches for successfully bringing applications to the market -especially in a well-defined domain - with high quality and low cost and time development is domain engineering. Its main goal is to support the systematic development of a set of similar systems by understanding and managing their common and variable features.

A domain engineering process is composed of three essential steps: domain analysis, domain design, and domain implementation. However, the existing processes present gaps and lack of details in these steps. Thus, in order to solve the problems found in the processes, this thesis proposed RiDE: the RiSE process for domain engineering. This Chapter presented its foundations, elements, and, briefly, each of its steps. The next Chapter will discuss in more details its first step: the domain analysis.

The Domain Analysis Step

"Academic organizations in computer science seem to prefer to work on new theories. However, some of the most successful academic work is a fusion and formalization of successful techniques"

James Neighbors (1989)
Domain Analysis Creator

Domain analysis has been identified as an important factor in the development of reusable software, since its philosophy is to analyze the systems collectively rather than separately, and identify common and variable features. Although it is not a new idea, for domain analysis to become a practical approach it is necessary, among other things, to understand the conceptual foundations of the process and to produce an unambiguous definition in the form of specific techniques.

In this sense, this Chapter presents the domain analysis approach, which is the first step of RiDE, its principles, activities, sub-activities, and roles. The approach is based on the strong and weak points of the reuse processes discussed in Chapter 3 in conjunction with other relevant directions in the area.

5.1. Introduction

The term domain analysis was first introduced by Neighbors (1980) as "*the activity of identifying the objects and operations of a class of similar systems in a particular problem domain.*" However, neither Neighbors' nor many other works (Arango, 1989), (Prieto-Diaz, 1990), (Almeida et al., 2005a) address the issue of "*how to perform*" domain analysis. These works focus on the outcome,

not on the process. This issue can also be observed in the software reuse processes, which present gaps in the domain analysis activities, as discussed in the previous Chapter.

Thus, improvements related to domain analysis are necessary in the reuse processes, which usually conduct it in an ad-hoc manner and success stories are more exceptions than rules. The process of concept abstraction, from the identification of common features, is usually considered as an exclusive human activity and commonly associated with experience. However, little is known about the process involved in deriving and organizing such collections of abstract concepts. Gaining experience is a slow unstructured learning process. Similarly, domain analysis is a slow unstructured learning process that leads to the identification, abstraction, and encapsulation of objects in a particular domain (Prieto-Diaz, 1990).

Typically, knowledge of a domain evolves over time until enough experience has been accumulated and several systems have been implemented, so generic abstractions can be isolated and reused. In domain analysis, experience and knowledge are accumulated until it reaches a threshold. This threshold can be defined as the point when an abstraction can be organized and made available for reuse.

Thus, in order to truly exploit reusability in industrial environments it is necessary to develop systematic ways to perform domain analysis. In this context, our goal with domain analysis, in concordance with Prieto-Diaz (1990), (pg. 48) is: "*to find ways to extract, organize, represent, manipulate and understand reusable information, to formalize the domain analysis process, and to develop technologies and tools to support it*."

Our research is based on the following assumptions: **i.** problem domains exist; **ii.** problem domains evolve gradually; **iii.** there are organizations that need to develop large numbers of similar systems within those domains; **iv.** there is expertise in building systems in those domains; and, finally, **v.** reusers follow systematic approaches to reuse. Thus, the focus is on an approach which aims to (Almeida et al., 2006): understand the conceptual foundations of the domain analysis process; produce an unambiguous definition in the form of specific techniques; and define a common notation for the process. Moreover,

the approach should present a systematic set of principles, guidelines, metrics, roles, inputs and outputs.

5.2. The Principles

In order to obtain a practical and effective way to perform domain analysis, the proposed approach is based on a set of Analysis Principles (AP):

AP_1. Scoping: project management activities, for domain engineering and product lines are sometimes called product line scoping. The main goal of scoping methods and approaches is to identify the products that will belong to the product lines as well as to define their major features.

AP_2. Metrics: metrics should be used whenever applicable, although not all engineering activities can be carried out with metrics. So, if metrics are not applicable, the approach should provide guidelines rather than simply giving general principles.

AP_3. Flexibility: features defined in the feature model can be used to parameterize domain architectures and components in the future. Thus, components can be developed almost free of design decisions by putting the features in the components as instantiation parameters.

AP_4. Commonality analysis: the goal of commonality analysis is to identify which features (requirements) are common to all applications of the domain.

AP_5. Variability analysis: the goal of variability analysis is to identify which features (requirements) differ among the applications, and to determine the differences precisely.

AP_6. Variability Modeling: this modeling concerns with modeling variation points, variants, and their relationships.

AP_7. Traceability: traceability links, when well documented, can ensure the consistent definition of the commonality and the variability of the domain throughout all assets specified.

AP_8. Validation and documentation: one important consideration in any method or process is related to activities for validation and documentation of

the developed assets, mainly, in the context of domains and product lines, where asset complexity and volume are enormous.

AP_9. Systematic sequence of activities: the last principle and not less important, is that a systematic sequence of activities must be included, with the ordering of activities being logical and easy to apply in practice.

In this context, the main goal of domain analysis is to identify commonalities and variability of applications in a domain and represent them in an effective form. The approach for domain analysis consists of three activities, as shown in Figure 5.1, using SA language (Ross, 1977): **Plan Domain, Model Domain** and **Validate Domain**. The next Sections present each activity in details. In order to aid the understanding of the inputs, outputs, and controls, the starship game domain will be used.

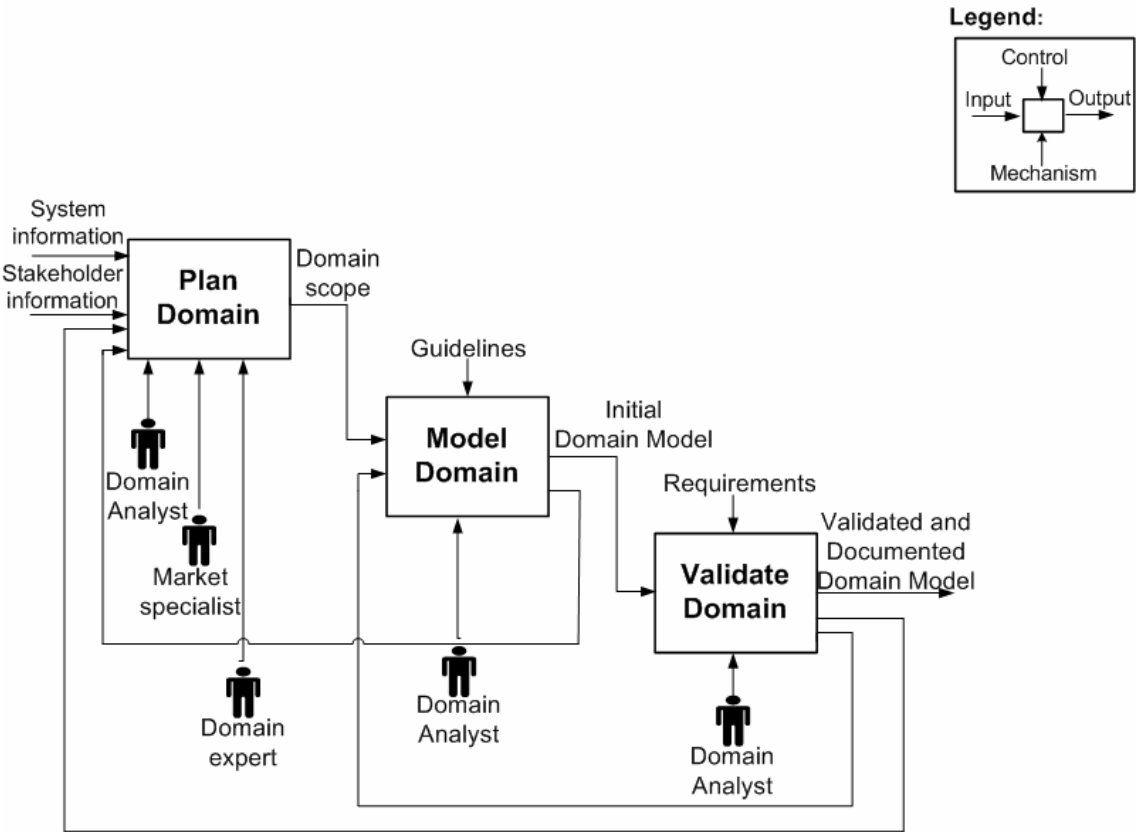

Figure 5.1. Domain Analysis Activities.

5.3. Plan Domain

The *first activity* in the approach corresponds to a preparation phase. One of the goals in this activity is to determine whether it makes good sense to invest in building a reuse infrastructure in a given domain.

Initially, the domain analyst – a person who conducts the domain analysis process – based on the chosen domain, performs the following sub-activities:

- **Analyze Stakeholders**: encompasses the identification of the stakeholders and their roles within the process. A stakeholder is someone who has a defined interest in the outcome of the project;
- **Define Objectives**: corresponds to the definition of the stakeholder's desired objectives for the project;
- **Define Constraints**: comprehends the definition of the restrictions imposed by the organization and by market conditions, making clear what is beyond the process scope;
- **Analyze Market (if applicable)**: this non-trivial sub-activity – which can be performed or not, depending on costs, complexity and maturity of the organization in the domain – is the systematic research and analysis of the external factors that determine the success of the domain in the marketplace. It involves the gathering of business intelligence, competitive studies and assessments, market segmentation, customer plans, and the integration of this information into a cohesive business strategy and plan (Clements & Northrop, 2001); and
- **Collect Data**: encompasses sub-activities to elicit and examine existing knowledge about the potential domain in focus. It includes the analysis of all available documentation (project plans, user manuals, modeling, and data dictionary), existing applications and knowledge from domain experts.

It is important to highlight that these sub-activities are not sequential and can be performed in parallel. The template presented in Appendix D is used

to document these sub-activities. After performing the above sub-activities, the domain analyst starts to identify the features and to define the domain scope. These tasks are organized in four sub-activities, as shows Figure 5.2. The next Sections present each sub-activity in detail.

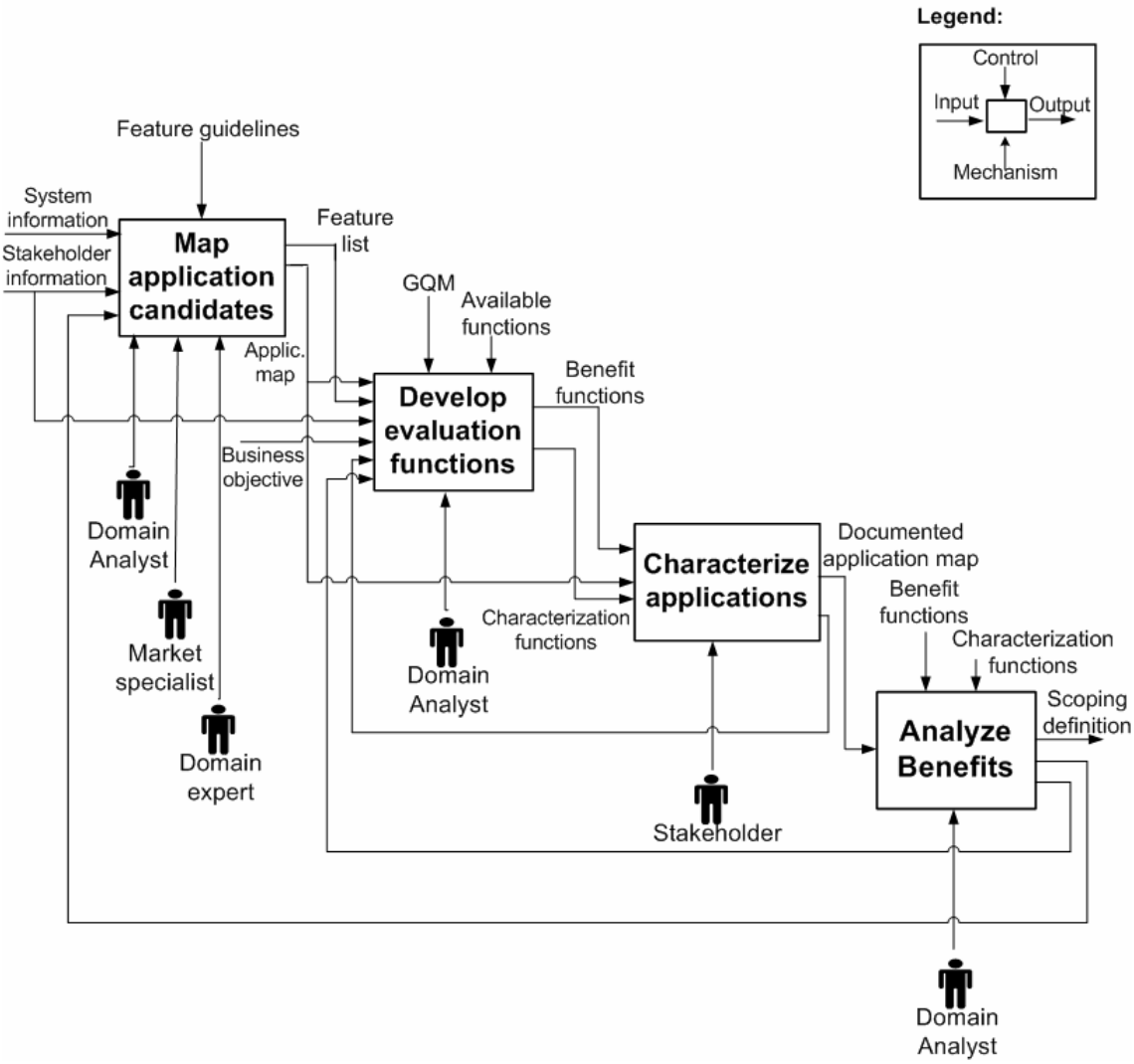

Figure 5.2. Domain Scope Sub-Activities.

5.3.1. Map application candidates

This initial sub-activity aims at identifying the characteristics that shall be supported by the future domain architecture. Thus, firstly, the domain analyst, based on system and stakeholder information, identifies the applications that might be supported by the domain sequentially.

Three types of applications can be distinguished: *existing applications* (i.e., applications that have been developed prior to the start of the domain

analysis process), *future applications* (i.e., applications where the requirements are rather clear, but development has not yet started) and *potential applications* (i.e., applications for which no clear requirements exist yet, but these applications are seen as relevant).

After understanding the relevant applications, the *list of features* is developed. The use of features is motivated by the fact that customers and engineers often speak of application characteristics in terms of features that the application has and/or delivers (Kang et al., 1990). This list provides a description of the different characteristics and identifies relations among them.

Identification of features involves abstracting domain knowledge obtained from the domain experts and other documents such as books, user manuals, design documents, and source programs (Lee et al., 2002). However, the volume of documents and information to be analyzed tends to be too big in a domain. In this context, the Analysis Guidelines (AG) defined initially by Lee* et al. (2002) (AG1 and AG3) can be very useful:

AG_1. Analyze terminologies used in the domain to identify features. In some mature domains, experts normally use standardized terminology to communicate their ideas, needs, and problems. Thus, using standard terms for the feature identification can expedite communication between domain analyst and information providers (domain experts and end-users).

Lee et al. categorize the features in a feature identification framework. This framework organizes the features in four groups: *capability*, *domain technology*, *implementation techniques*, and *operating environments*.

Capability features are characterized as distinct services, operations or non-functional aspects of applications in a domain. Services are end-user visible functionality of products offered to their users in order to satisfy their requirements. Operations are internal functions of applications that are needed to provide services. Non-functional features include end-user visible application characteristics that cannot be identified in terms of services or operations, such as presentation, capacity, quality attributes, etc.

* The guidelines originally defined by Lee et al. were proposed for software product lines. We adapted them to work with the domain vision.

Domain technology (e.g., movement methods in a game domain) features are domain-specific technologies that the domain analyst or architect uses to model specific problems or to implement service or operation features. These features are specific to a given domain and may not be useful in other domains.

Implementation techniques features are more generic than domain technology features and may be used in different domains. They contain key design and implementation decisions that may be used to implement other features. Some examples are: design patterns and architectural styles.

Operating environments (e.g., operating systems) features include domain contexts, such as computing environments and interfaces with different types of devices and external systems.

We believe that organizations starting to explore domain analysis in their projects, or in immature or unstable domains, should first concentrate just on capability features. Next, after achieving improvements in domain knowledge and analysis, the other kinds of features can also be used.

AG_2. Try to first find differences among existing applications in a domain and then, with this understanding, identify commonalities. Applications in the same domain share a high level of commonality (Coplin et al., 1998). Hence, the commonality space would be larger to work with than the difference space, thus finding difference is much easier than finding commonalities. So, the strategy is, initially, to identify the existing applications (running systems, documentation, etc), and then list different features that characterize each one. After this understanding, the identification of common features is easier to be performed.

AG_3. Do not identify all implementation details that do not distinguish between applications in a domain. A developer tends to enumerate all the implementation details and to identify them as features, even though there are no variations among them. But it is important to note that a feature model is not as detailed as a requirement model, where desired functionality is expressed for developers to understand and implement it. For this reason, it is recommended

that this sub-activity should be performed by a domain analyst, who will abstract implementation details.

At the end, the applications and features information are organized in the form of an *application map* as shown in Figure 5.3. In this map, rows and columns are used to represent features and applications, respectively. Sometimes, it is common to divide a feature in a set of sub-features, when there is a relationship between them; for example, the feature *Actions*, in the starship game domain, can be divided in sub-features, such as *shooting, recompose, divide,* etc. The template presented in Appendix C is used to document this sub-activity. This domain is composed of classic arcade games such as Space Invaders, Demon Attack and Phoenix where the player has to destroy all the enemies which are starships and obstacles presented for him.

				SPACE INVADERS				DEMON ATTACK				PHOENIX			
ID Feat.		ID Feat.		Req	Sim	Eff(c, a)	Eff(c)	Req	Sim	Eff(c, a)	Eff(c)	Req	Sim	Eff(c, a)	Eff(c)
[F 01.01.01.01]	Obstacle														
[F 01.01.01.02]	Projectile														
[F 01.01.02.01]	Enemies														
[F 01.01.02.02]	Main Character														
		[F 01.01.02.02a]	Score												
[F 01.01.02.03]	Actions	[F 01.01.02.03a]	Shooting												
		[F 01.01.02.03b]	Recompose												
		[F 01.01.02.03c]	Divide												
		[F 01.01.02.03d]	Shield												

Figure 5.3. Map Applications Candidate Sub-Activity.

5.3.2. Develop evaluation functions

In this sub-activity, based on the information that was previously produced, the domain analyst refines and operationalizes the business objectives that are relevant to the domain in the particular context.

This refinement is important because initially (in the previous sub-activity) the objectives are identified in a manner that is too generic for being directly used. Thus, the approach includes the concept of successively refining the business objectives towards more operational evaluation criteria, as also used in the PuLSE methodology (Bayer et al., 1999). These criteria can be applied on a single application/feature combination. This refinement has the

advantage that an explicit and traceable relation between the business objectives and the identified scope is established.

In this context, this sub-activity starts when the domain analyst identifies the relevant stakeholders and elicits their objectives. Next, in order to make the goal more precise, a goal schema based on Goal Question Metric (GQM) (Basili et al., 1994) is used. The schema has the form of <purpose><Issue><Object><Context>, such as ***Minimize*** *the* ***effort*** *needed for* ***the development of new applications*** *from the viewpoint of* ***software engineers*** *in the* ***company***.

After defining the goal, questions are used to elicit additional information about the objectives in order to capture them in a more effective way. Based on the objectives and questions, the domain analyst develops or reuses *evaluation functions*, which produces *benefit* and *characterization functions*. The *benefit functions* describe the benefit of introducing a certain feature into the reuse infrastructure relative to a certain business objective. The *characterization functions* are applied on single applications/features combinations and play a role analogous to metrics in the GQM approach. Table 5.1 presents two results of this sub-activity (Develop evaluation functions): the first, in the component repository domain, and the second one, in a generic domain.

Table 5.1. Develop Evaluation Functions.

Objective	*To attend most of the domain applications, maximizing the repository use and standardizing the characteristics that are available in the applications*
Questions	*1. How to define if the repository support or not an application?* *2. How to evaluate the level of use from the repository by the applications?* *3. How to determinate the deviation of a characteristic of an application in relation to the established pattern?*
Characterization	• *Is characteristic* c *necessary for the application* a*?* - req(c,a) – *1:yes; 0:no* • *Similarity of the characteristic* c_a *with the standard characteristic* c *in the application* a - sim(c,c_a,a) – *1:equal the standard; 0.5:next to the standard; 0:different of the standard*
Benefit	• *Proximity of the characteristic in the applications in relation with the standard* - P(c,a) – $\sum_{req(c,a)}$ x $\sum_{sim(c,ca,a)}$ • *Distance of the characteristic in the applications in relation with the standard* - D(c) – $\sum_{req(c,a)}$ x (1 – sim(c,c_a,a))

Objective	*To allow the effort reduction for new application development in a domain*
Questions	*1. Is the characteristic important for the domain?* *2. Does the characteristic have an important ROI?*
Characterization	• *Effort to include the characteristic* c *in the application* a - eff(c,a) – *man/hour* • *Effort to implement the characteristic* c - eff(c) – *man/hour*
Benefit	• *Effort economy to develop the domain applications reusing the standard implementation of the characteristic* c - E(c) – $\sum_{req(c,a)}$ x (eff(c_a) – eff(c,a)) • *Effort to develop the standard implementation of the characteristic* c - E(c) – eff(c) x (1 + D(c))

5.3.3. Characterize Applications

In this sub-activity, the characterization functions are applied to the *application map*. Usually this is done by stakeholders who provided the different objectives (e.g., the project leader for development can be useful to estimate effort, while a marketing expert can be useful to describe which features may be useful to gain market share), or other qualified person. It is, basically, a knowledge elicitation task centered on the characterization functions.

As can be seen in Figure 5.2, difficulties that were found during knowledge elicitation for the characterization functions lead to modifications in the application map and the evaluation functions. Moreover, the scales used in the functions can also be changed. The information resulting from these iterations is entered into the application map and leads to the *documented application map* as shown in Figure 5.4.

				SPACE INVADERS				DEMON ATTACK				PHOENIX			
ID Feat.		ID Feat.		Req	Sim	Eff(c, a)	Eff(c)	Req	Sim	Eff(c, a)	Eff(c)	Req	Sim	Eff(c, a)	Eff(c)
[F 01.01.01.01]	Obstacle														
[F 01.01.01.02]	Projectile														
[F 01.01.02.01]	Enemies														
[F 01.01.02.02]	Main Character														
		[F 01.01.02.02a]	Score												
[F 01.01.02.03]	Actions	[F 01.01.02.03a]	Shooting												
		[F 01.01.02.03b]	Recompose												
		[F 01.01.02.03c]	Divide	0,0	0,0	0,0	0,0	1,0	1,0	0,3	6,8	0,0	0,0	0,0	0,0
		[F 01.01.02.03d]	Shield												

Figure 5.4. Documented Application Map.

According to the Figure, the values are computed in the following way, considering just the feature *Divide*:

- `req (c, a)` = 0, in Space Invaders and Phoenix games, since this feature is not present in these applications;
- `req (c, a)` = 1, in Demon Attack, since this feature is present in this game;
- `sim (c, ca, a)` = 0, in Space Invaders and Phoenix games, since this feature is not present in these applications;
- `sim (c, ca, a)` = 1, in Demon Attack, since the goal is to design this feature according to the standard used in this game;
- `eff (c, a)` = 0, in Space Invaders and Phoenix games, since these games do not have this feature;
- `eff (c, a)` = 0.3, in Demon Attack, it being an estimative performed by the stakeholder;

- `eff (c)` = 0, in Space Invaders and Phoenix games, since these games do not have this feature; and
- `eff (c) = 6.8`, in Demon Attack, given an estimative by the stakeholder.

5.3.4. Analyze Benefits

The last sub-activity of the domain planning is very important. At this point the information acquired is used to derive the *scope definition* (Appendix C presents the template for Domain Scoping Activities). The first step is to identify the adequate values for the benefit functions. This is done by applying the definition functions to the values elicited for the characterization functions in the previous sub-activity.

One important warning is related to differences in scale. Some scales that are used for the characterization functions can be nominal or ordinal, thus, its definition should take this point in consideration, in order to allow arithmetic operations to be performed on these values.

After values were assigned to the benefit functions, it is possible to determine the scope that is aligned with the business objective. Some multi-objective decision techniques, described in literature, can be useful for this proposal (Mollaghasemi & Pet-Edwards, 1997). In RiDE, we use a weighting scheme to perform it.

Once candidate features are identified based on the described sub-activities, these features should be organized into a domain model. The details of this activity are described in the next Section.

5.4. Model Domain

In the *Plan Domain* activity, a domain was selected based on a strategic analysis of the stakeholders' interest on the process, which ensured a sufficient business case to develop a *domain model* for the selected domain. The purpose of the *Model Domain* activity is to fill in the content within the domain itself. This means a shift of attention from scoping issues to structural issues and conceptual elements within the domain.

Thus, the resulting model should describe the commonality and variability within the domain. Rather than building a model for a single application in the domain, or even a generic model that may be applicable at a high level to a number of applications, the domain modeling task attempts to formalize the space of variations for individual applications in the domain.

In this approach, the domain model is represented through *feature models* (Kang et al., 1990). The domain analyst groups the features that were identified in the previous step in a features model, a graphical AND/OR hierarchy of features, that captures logical structural relationships (e.g., composition and generalization) among features. Three types of relationships are represented in this diagram: "composed-of", "generalization/specialization", and "implemented by" (Kang et al., 1998). Features may be *mandatory*, *optional*, *or features* or *alternative* as shows Figure 5.5 (Czarnecki & Eisenecker, 2000).

A *mandatory feature* node is pointed to by a simple edge ending with a filled circle. According to Figure 5.5, every instance of concept C has features f_1 and f_2, and every instance of C that has f_1, has f_3 and f_4. Thus, every instance of C has f_3 and f_4. In summary, every instance of C can be described by the feature set {C, f_1, f_2, f_3, f_4}.

An *optional feature* may be included in the description of a concept instance if and only if its parent is included in the description. Thus, if the parent is included, the optional feature may be included or not, and if the parent is not included, the optional feature cannot be included. According to the Figure, an instance of C might have one of the following descriptions: {C}, {C, f_1}, {C, f_1, f_3}, {C, f_2}, {C, f_1, f_2}, or {C, f_1, f_2, f_3}.

A Concept may have one or more sets of direct *alternative features*. Similarly, a feature may have one or more sets of direct alternative sub-features. If the parent of a set of alternative features is included in the description of a concept instance, then exactly one feature from this set of alternative features is included in the description; otherwise none are included. According to the Figure, an instance of C can derive the following instance descriptions: {C, f_1, f_3}, {C, f_1, f_4}, {C, f_1, f_5}, {C, f_2, f_3}, {C, f_2, f_4}, or {C, f_2, f_5}.

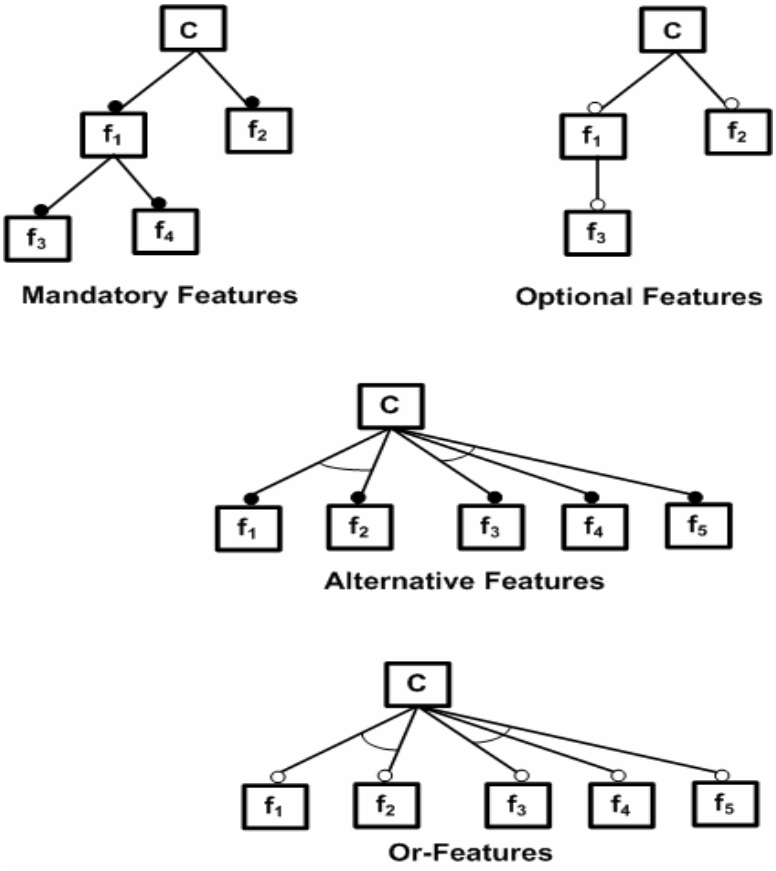

Figure 5.5. Feature Types.

A concept may have one or more sets of direct *or-features*. Similarly, a feature may have one or more sets of direct or-sub-features. If the parent of a set of or-feature is included in the description of a concept instance, then any nonempty subset from the set of or-features is included in the description; otherwise, none are included. According to the Figure, `C` has two sets of *or-features*: one set with f_1 and f_2 and another set with f_3, f_4, and f_5. A total of $(2^2 - 1)$ $(2^3 - 1)$ or 21 different instance descriptions may be derived from this diagram.

Figure 5.6 shows part of the feature model from the starship game domain with its feature types. According to the Figure: *all* game has a `flow` and `entity` (mandatory features); in *some* games, the player can win special `bonus`, for example, after establishing a new score or accumulating a specific score (optional feature); all `entity` have `movement` and this one has a `direction` what can be *or* `horizontal` or `vertical` (alternative features);

and, finally, in some games, the `movement` can have a `speed` and/or `acceleration` (or-features).

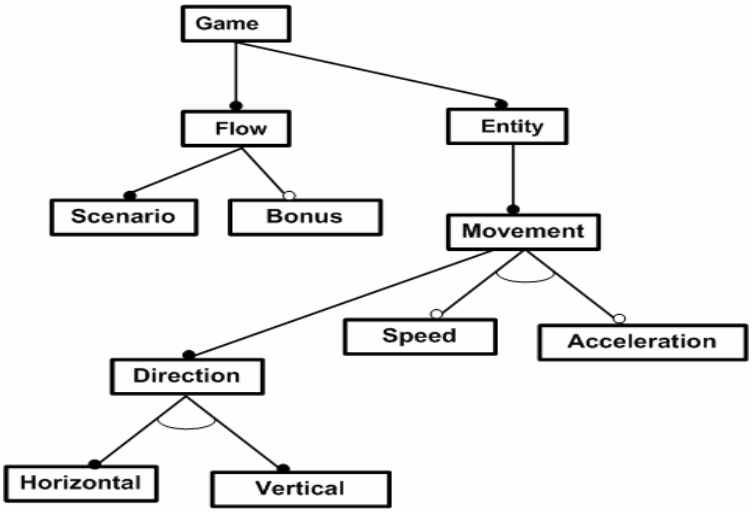

Figure 5.6. Feature Types in the Starship Game Domain.

Composition rules supplement the feature model with information showing mutual dependency and mutual exclusion relationships between variant features and variation points. It is used to verify the consistency and completeness of the selected features. Variation points and variants are used to define the variability of a domain. According to Pohl et al. (2005), a *variation point is a representation of a variability subject* – a variable item of the real world or a variable property of such an item - *within domain assets enriched by contextual information*. On the other hand, a *variant is a representation of a variability object within domain assets*. In our approach, three types of relationships (R) are considered:

R_1. Variant Constraint Dependency. A variant constraint dependency describes a relationship between two variants, which may be one of two types:

- **Variant requires variant**. The selection of a variant V_1 requires the selection of another variant V_2 independent on the variation points the variants are associated with.
- **Variant excludes variant**. The selection of a variant V_1 excludes the selection of the related variant V_2 independent on the variation points the variants are associated with.

Figure 5.7 shows the variant constraint dependency relationship. As the starship game domain does not have all the three types of relationships in its feature model, generic names[1] are used to represent the variant and variation points.

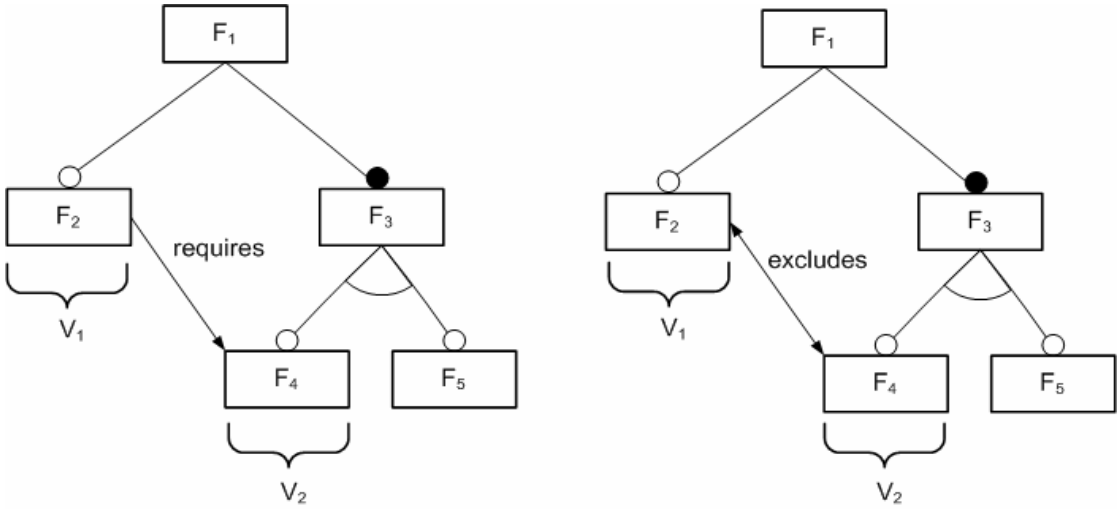

Figure 5.7. Variant Constraint Dependency Relationship.

R_2. Variant to Variation Point Constraint Dependency. The variant to variation point constraint dependency describes a relationship between a variant and a variation point, which may be one of two types:

- **Variant requires variation point**. The selection of a variant V_1 requires the consideration of a variant point VP_2.
- **Variant excludes variation point**. The selection of a variant V_1 excludes the consideration of a variation point VP_2.

Figure 5.8 shows the variant to variation point constraint dependency relationship.

[1] The letter "F" was chosen because the variants and variation points represent features in the semantic sense.

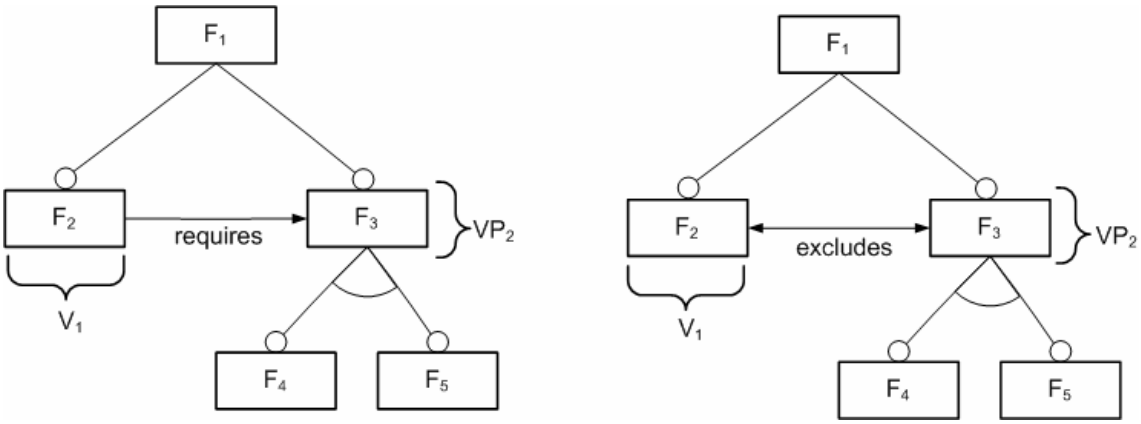

Figure 5.8. Variant to Variation Point Constraint Dependency Relationship.

R_3. Variation Point Constraint Dependency. A variation point constraint dependency describes a relationship between two variation points, which may be one of two types:

- **Variation point requires variation point**. A variation point requires the consideration of another variant point in order to be realized.
- **Variation point excludes variation point**. The consideration of a variation point excludes the consideration of another variation point.

Figure 5.9 shows the variation point constraint dependency relationship.

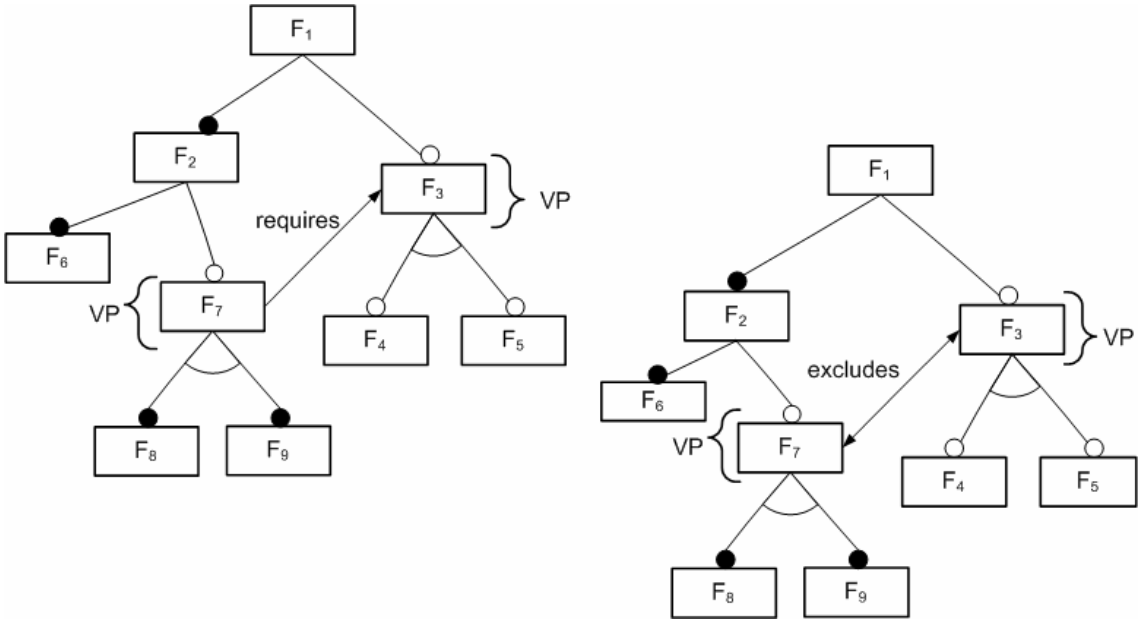

Figure 5.9. Variation Point Constraint Dependency.

Lee et al. (2002) defines two useful analysis guidelines that can be used to organize features:

AG_4. Do not organize features to represent functional dependencies, like a function call hierarchy, but organize features to capture and represent commonalities and differences. Developers who are familiar with a structured development method often confuse a feature model with a functional call hierarchy. Thus they tend to identify all functions of applications as features and organize them similar to a functional call hierarchy. However, features should be organized so that commonalities and variabilities can be easily reorganized, rather than representing interactions among features, like a function call hierarchy.

AG_5. If many upper-level features are associated with a lower-level feature through the implemented-by relationship, reduce complexity of the feature model by associating the lower-level feature with the nearest common parent of its related upper-level features. When upper-level features, in a feature model, are associated with several lower-level features through the implementation-by relationship, the implementation relationships between those features may be very complex. Since the primary concern of feature modeling is to represent commonality and variability in a domain rather than to model implementation relationships between features, complex relationships can be reduced by associating the lower-level feature with the nearest common parent of its related upper-level features. This guideline is useful in mature or stable domains where domain and technology features were identified. But in domains where just capability features were identified it is not applied.

Besides these guidelines, proposed by Lee et al., an important consideration (C) related to feature models must be considered:

C_1. A feature model with AND-nodes at an upper level and OR-nodes at a lower level usually indicates a high level of reuse opportunity. On the other hand*, alternatives (i.e., OR-nodes) at the upper level usually mean that applications in the domain do not share much commonality in terms of services and functions provided by them*. This indicates that the domain might not have much reuse opportunity at the application level, although there might still be opportunities for reuse at low level. Additionally, *alternatives (OR-*

nodes) at a lower level indicate different ways of designing and implementing certain reusable information.

Once feature models are organized in a coherent structure composed of features, common and variable points, relationships, and composition rules, the domain analyst performs the domain validation.

5.5. Validate Domain

Before the domain model is ready for use, it is necessary to validate and document it; however, there are not many works in this direction. Addy (1998) presents a framework that extends verification and validation from an individual application system to a product line of systems that are developed within an architecture-based software engineering environment. Nevertheless, Addy's work is very generic and does not discuss its task systematically, which makes difficult its practical usage.

Thus, in our approach, in order to achieve the domain validation, the domain analyst performs the following sub-activities (A):

A_1. Document features. For this sub-activity, the template defined by Czarnecki & Eisenecker (2000) is used. In this template each feature consists of:

- **Semantic description.** Each feature should have at least a short description describing its semantics;
- **Rationale.** A feature should have a note explaining why the feature is included in the model;
- **Stakeholders and client programs.** Each feature should be annotated with stakeholders (e.g., users, customers, developers, managers) who are interested in the feature and the client programs that need this feature;
- **Example applications.** If possible, the documentation should describe features with known applications implementing them;
- **Constraints.** Constraints are hard dependencies between variable features. Two important kinds of constraints are mutual-exclusion constraints and required constraints.

- **Open/closed attribute.** Variation points should be marked as open if new direct variable sub-features (or features) are expected. On the other hand, marking a variation point as closed indicates that no other direct variable sub-features (or features) are expected; and
- **Priorities.** Priorities may be assigned to features in order to record their relevance to the process.

A_2. Check for synonyms. Once the semantic description has been defined, it is necessary to analyze each feature in order to find and eliminate synonyms, i.e., different terms that appear to have the same domain-relevant meaning;

A_3. Check for homonyms. As a complementary sub-activity to the search for synonyms, it is necessary to check each feature in order to find homonyms, i.e., the same literal term used with different meanings in different contexts;

A_4. Model Validation. After the features documentation, the check for synonyms and homonyms, the domain analyst performs the model validation. This sub-activity corresponds to the matching of the requirements that were expressed by stakeholders and the domain model, in order to validate its completeness and accuracy; and

A_5. Document the domain. The last sub-activity corresponds to the documentation of the domain. In order to do it, the meta-model defined in (Schmid et al., 2001) is used, consisting of the following information:

- **Domain description.** Defines the responsibilities of the domain;
- **Domain defining rules.** Includes decisions criteria about the inclusion and exclusion of domain membership and the logical relationships between these criteria;
- **Exemplar system selection.** Denotes a set of systems in the scope where the domain functionality occurs;
- **Documentation.** Describes a set of documents related to exemplar systems;

- **Domain Context (Relationship).** Describes the relation between the domain in focus and other domains;
- **Domain genealogy.** Encompasses information about the evolution and dependencies among systems within a domain; and
- **Feature.** Defines a set of features described in a domain.

Figure 5.10 shows the elements present in this meta-model. Appendix D presents in detail the template for domain documentation.

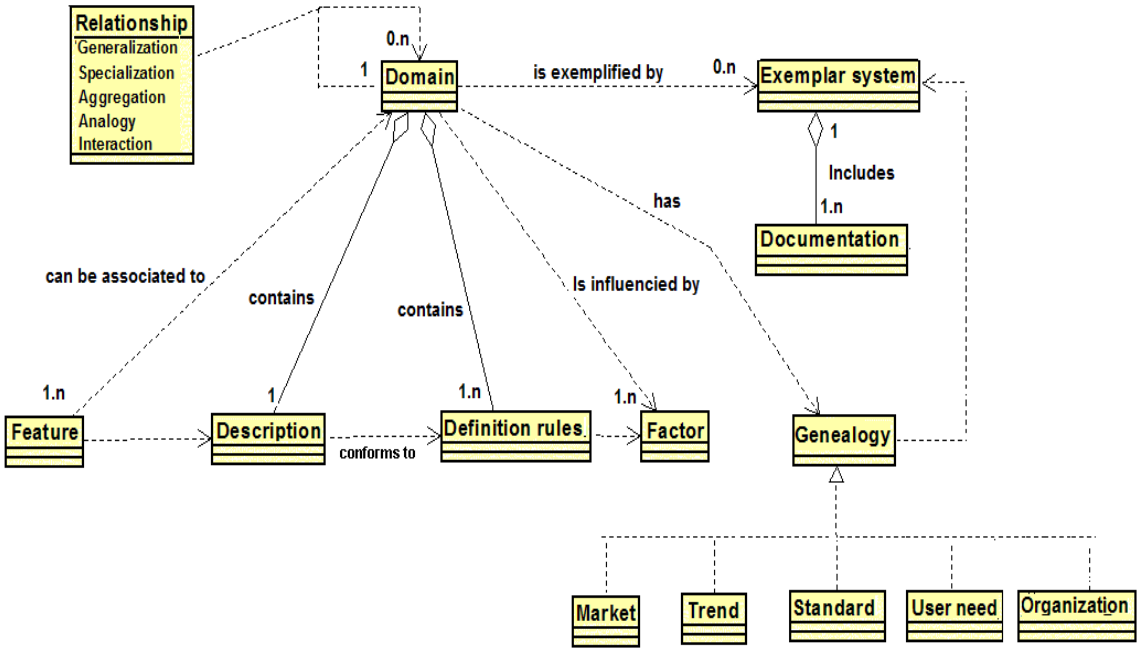

Figure 5.10. Elements in the Domain.

5.6. Summary

Table 5.2 presents a summary of the approach with Activities (A), Sub-Activities (S), Inputs (I), Outputs (O), and the Roles (R).

Table 5.2. Summary of the Domain Analysis Step.

A	S	I	O	R
Plan Domain	Stakeholder analysis	Domain, Project	Stakeholder list, Stakeholder roles	Domain analyst, Domain expert
	Objectives definition	Stakeholder list	Objectives	Domain analyst, Domain expert
	Constraint definition	Objectives	Project scope	Domain analyst
	Market analysis	Domain, Project, Constraints	Business plan	Market specialist, Domain expert
	Data collection	Documentation, Applications, Expert knowledge	Partial domain knowledge	Domain analyst, Domain expert
	Map application candidates	System and Stakeholder information	Feature list, Application map	Domain analyst
	Develop evaluation functions	Business objectives, App map, Stakeholder information	Benefit functions, Characterization functions, App map	Domain analyst
	Characterize applications	Benefit functions, Characterization functions, App map	Documented map	Stakeholder
	Analyze Benefits	Application map	Domain scope	Domain analyst
Model Domain	Feature modeling	Application map	Domain model, Composition rules	Domain analyst
Validate Domain	Document features	Domain model	Documented features	Domain analyst
	Check for synonyms	Documented features	Refined features	
	Check for homonyms	Documented features	Refined features	
	Model validation	Domain model, Requirements	Validated domain	
	Document the domain	Domain model	Documented domain	

5.7. Other Directions in Domain Analysis

The domain analysis area is not new and some results are already discussed in the literature. The works described in (Debaud & Schmid, 1999) and John et al. (2006) are close to the Domain Scope activity defined here, with the main difference that these are directed to software product lines. Besides, the work defined in John et al. is derived from industrial experience. In a certain sense, the software reuse processes (discussed in Chapter 3) served as an initial inspiration to the approach presented, mainly, in its weak and strong points. However, other directions were also analyzed, such as the works proposed by

Frakes et al. (1998), Bayer et al. (1999a), Kim et al. (2003), Mei et al. (2003), and Moon et al. (2005).

Frakes et al. define a method and a CASE tool for helping in the achievement of systematic reuse through domain analysis. The aspect discussed in their work is the use of a prototype to automate the domain analysis process. The tool is used to extract knowledge from documents and source code and organize it to be reused. But, on the other hand, the method does not present improvements in the domain analysis area, since it does not discuss, for example, how to perform scope and validation domain, or modeling activities.

Bayer et al. present the Customizable Domain Analysis (CDA) method, which is a subset of the PuLSE methodology. The CDA method was developed to be adapted to the project needs and provide guidance to be systematically applicable. It consists of three steps: refine scope definition, elicit raw domain knowledge, and model domain knowledge. The problem with this method is that its steps are very generic, lacking of details on how to perform it.

Kim et al. propose a new direction in domain analysis with the use of goals, scenarios, and features. Their approach aims at offering a systematic and concrete method for identifying features and providing the rationale for the features and the commonality and variability analysis. The method is interesting in the domain modeling task. However, the task of scoping and domain validation is not considered. Moreover, the method does not systematically define inputs, outputs and roles, as in a more effective domain analysis process.

Mei et al. define FODM, a Feature-Oriented Domain Modeling method, which explores FODA ideas. Moon et al. propose a process of producing domain requirements where commonality and variability are explicitly considered. These approaches are very interesting, mainly the second one. However, the work presented in this thesis is more complete due to the fact that it treats all the steps of domain analysis, which are not considered in Mei et al.'s and Moon et al.'s work. The first concerns just domain modeling and the second does not treat planning and validation.

5.8. Chapter Summary

Domain analysis is a key aspect for organizations to maximize the benefits with software reuse, mainly, in a high abstraction level. However, the existing approaches present more emphasis in some activities of the process such as scoping or modeling, while other relevant ones treat it in an ad-hoc way, increasing the risks, since in domain analysis the concern is a set of applications in a domain instead of just one application.

In this context, this Chapter discussed the first step of RiDE, the domain analysis, based on a set of guidelines, metrics, roles, inputs and outputs. The next Chapter will present the second step of the process: the domain design, in which the main goal is to design the domain-specific software architecture in order to support several applications in the domain.

The Domain Design Step

"We consider a set of programs to constitute a family, whenever it is worthwhile to study programs from the set by first studying the common properties of the set and then determining the special properties of the individual family members"

David Lorge Parnas (1976)
Software Product Lines Precursor

As presented in Chapter 4, the domain engineering life cycle encompasses three important steps: domain analysis, domain design and domain implementation. Although all the steps are important, the **domain design** stands out as crucial (Bosch, 2000), since its general purpose is to produce a Domain-Specific Software Architecture (DSSA), i.e. the common architecture for a set of applications. This task is even more relevant in the context of a domain engineering process, because all the output specified in the domain analysis should be mapped to the design specifications. Thus, the domain architect should determine systematically how requirements, including variability, are reflected in the architecture.

This Chapter presents the domain design approach, which is the second step of RiDE, its principles, guidelines, activities, sub-activities, and roles. The approach is based on the strong and weak points of the reuse processes discussed in Chapter 3 in conjunction with other relevant directions in the software architecture and component-based development area.

6.1. Introduction

A critical aspect of the design for any large software system is its high-level organization of computational elements and interactions between those elements, i.e., the software architecture level of design (Garlan, 1995).

According to Kruchten et al. (2006), the first reference to the phrase software architecture occurred in 1969 at a conference on software engineering techniques organized by NATO. Some pioneers, such as Edsger Dijkstra, Alan Perlis, and Niklaus Wirth attended the meeting.

From then until the late 1980s, the word *architecture* was used mostly in the sense of system architecture (meaning a computer system's physical structure) or sometimes in the narrower sense of a given family of computers' instruction set (Kruchten et al., 2006).

The concept of software architecture as a distinct discipline started to emerge in 1990, and in 1995 (Shaw & Garlan, 1996), the field had a strong growth with contributions from industry and academia, such as methods (Kazman et al., 1994), approaches (Kruchten, 1995), and design patterns (Gamma et al., 1995) for software architecture. At the same time, large software development organizations – mainly in systems, aerospace, and telecommunications - such as Lockheed Martin (Tracz, 1995), Nokia (Ran, 2000), Philips (Muller, 1995), Siemens (Hofmeister et al., 1999), and others started to pay attention to software architecture, in conjunction with the reuse community, to investigate ways of reusing software architectures (Kruchten et al., 2006).

The main way of reusing a software architecture is to design a Domain-Specific Software Architecture[1] (DSSA) (Tracz, 1995) or Product-Line Architecture[2] (Dikel et al., 1997). The difference between software architecture in general and a DSSA is that a DSSA is used by all applications in the domain. Thus, such architecture should be robust, flexible, and highly customizable to facilitate the instantiation of the core architecture in multiple applications.

[1] Term used by the reuse community and adopted in this thesis.
[2] Term used by the software product lines community. Both represent the same idea.

In the domain engineering life-cycle, a DSSA is defined in the domain design step. The key goal of the domain design is to produce the domain-specific or reference architecture, defining its main software elements and their interconnections (Bosch, 2000). The elements determine the static and dynamic decomposition that is valid for all applications in the domain. According to Tracz (1995), a DSSA is defined *as an assemblage of software components, specialized for a particular type of task (domain), generalized for effective use across that domain, composed in a standardized structure effective for building successful applications*. An important characteristic of this architecture is the ability to select and configure reusable software assets.

However, even being pointed out since the late 1980's as an important issue for reusing software (Perry & Wolf, 1992), the software architecture field, with its design methods, does not address all the particularities of DSSA design. This problem is also identified in software reuse processes, which present gaps in important tasks such as the identification, specification, and software components design.

The problem is even bigger when domain design works together with domain analysis, where the output of the second is used as input for the first. The same vision is also shared by the software reuse community (Frakes & Kang, 2005) pp. 534: "*There is also a need for a seamless integration between the models output from domain analysis and the inputs needed for domain implementations such as components, domain specific languages and applications generators*".

In this context, the next Sections present the second step of RiDE: domain design; based on a coherent set of principles, guidelines, metrics, roles, inputs and outputs.

6.2. The Principles

In order to obtain a systematic way to perform the domain design, the approach is based on a set of Design Principles (DP).

DP_1. Separation of Concerns and information hiding: using modules to represent and describe the architecture, the domain architect can separate different concerns and design effective ways for them to intercommunicate.

DP_2. Parameterization: features defined in the feature model are used to parameterize the architecture and components. Thus, components can be specified in a way that is free of design decisions, by embedding the features in the components as parameters.

DP_3. Components isolated from the connection mechanism: the way components are designed and specified in the approach is free of particular communication techniques, for example, distributed or centralized.

DP_4. Consistency: assets created in previous steps may be used in next steps. Thus, assets developed in domain analysis should be used in domain design in a consistent way.

DP_5. Metrics: metrics should be used whenever applicable, although not all the engineering activities can be carried out with metrics. So, if metrics are not applicable, the approach should provide guidelines to measure the reuse progress, rather than simply give general principles.

DP_6. Commonality and Variability: the approach should allow commonality and variability, modeled in domain analysis, to be represented in DSSA design, using pre-defined guidelines.

DP_7. Traceability: traceability links, when well documented, can ensure the consistent definition of the commonality and the variability of the domain throughout all assets specified.

DP_8. Systematic sequence of activities: the last principle, and not less important, is to include a systematic sequence of activities, in an order that is logical and easy to apply in practice.

The domain design approach (Almeida et al., 2007a) consists of five activities, as shows Figure 6.1, using the SA language: **Decompose Module**, **Refine Module**, **Represent Variability**, **Define Component** and **Represent Domain Architecture**. There are relationships among the activities which are not demonstrated in the Figure for the sake of legibility. The

next Sections present each activity in details. The approach is influenced by some approaches described in the literature, such as Attribute-Driven Design (ADD) (Bass et al., 2003), UML Components (Cheesman & Daniels, 2000), and the weak and strong points from the reuse processes presented in Chapter 3.

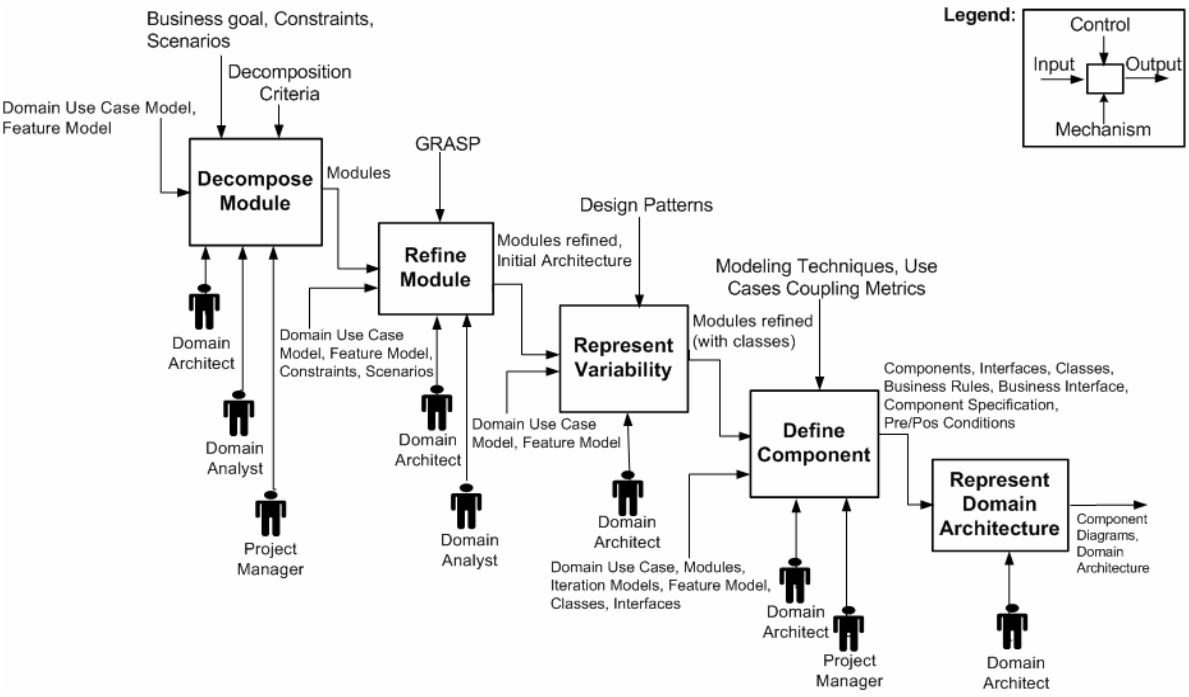

Figure 6.1. Domain Design's Activities.

6.3. Decompose Module

The *first activity* in the approach corresponds to an abstraction and decomposition phase. Initially, the domain architect – an experienced person who conducts the domain design process – based on the assets produced in the domain analysis (business goals, constraints, domain use case model, feature model, and scenarios), chooses the domain architecture modules to decompose. The modules to start are usually the whole applications in the domain, which are further decomposed into subsystems, and sub-modules. According to Parnas (1972), the benefits expected of modular decomposition are: **i. managerial** – development time should be shortened because separate groups can work on each module with little need for communication; **ii. flexibility** – it should be possible to make drastic changes to one module without a need for change in others; **iii. comprehensibility** – it should be possible to study the system one module at a time.

In this activity, the domain architect interacts with the domain analyst, in order to obtain detailed information about domain assets (mainly domain use case model, feature model and scenarios), and the project manager, to discuss the business goals and constraints that may have influence on the architectural design.

In the approach presented in this thesis, there is not a set of criteria to be used in module decomposition yet, such as Parnas' work, nor a set of rule of thumbs. However, based on the state-of-the-art and practical experience, we consider that the following issues should be balanced by the domain architect: *availability*, *coupling*, *extensibility*, *flexibility*, *functionality*, *information hiding*, *maintainability*, *modifiability*, *performance*, *separation of concerns*, *scalability*, *security*, and *usability*.

Figure 6.2 shows the modules defined in the starship game domain: *controller*, *entity*, *event*, and *util*. The *controller* is responsible to manage game's flow and control the interaction among the modules. *Entity* is responsible to create and monitor the objects (starships, projectiles, obstacles). *Event* processes and triggers the actions and reactions of the objects. At the end, *util* manages the audio of the applications and represents graphically the objects.

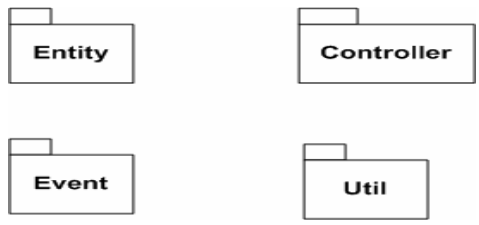

Figure 6.2. Modules defined in the Starship Game Domain.

After performing this activity, the domain architect performs the module refinement.

6.4. Refine Module

The module refinement is an iterative process that can be divided into three sub-activities: *Choose the Architectural Drivers*, *Choose the Architectural Patterns*, and *Allocate Functionality using Views*.

6.4.1. Choose the Architectural Drivers

According to Bass et al. (2003), *architectural drivers are the combination of functional and quality requirements that "shape" the architecture or the particular module under consideration*. The drivers are found among the top-priority requirements for the module.

The determination of architectural drivers is not always a top-down process. Sometimes detailed investigation is required in order to understand the ramifications of particular requirements. In RiDE, the module decomposition is based on the architectural drivers, because when choosing the drivers, we are reducing the problem to satisfy the most important ones. In the approach, the architectural drivers are the requirements expressed by the feature model, the quality attributes and the scenarios, if applicable.

6.4.2. Choose the Architectural Patterns

After identifying and choosing the relevant architectural drivers for each module, the domain architect selects the architectural patterns that can be applied. The patterns satisfy the architectural drivers and are constructed by composing selected tactics. Two factors guide tactic selection. The first is the drivers themselves, and the second is the side effects that a pattern implementing a tactic has on others. In our approach, the vision of a tactic agrees with Bass et al.'s, who define it as a design decision that influences the control of a quality attribute response. A tactic example can be one related to modifiability in order to prevent ripple effects, i.e., changes made in a module affecting other ones. Thus, some possible tactics can be information hiding, restrict communication paths, and the use of facades (Gamma et al., 1995).

6.4.3. Allocate Functionality using Views

In the previous sub-activities, the approach showed how the architectural drivers determine the decomposition structure of a module via the use of tactics. In this sub-activity, the goal is to define how the modules can be instantiated. The criteria for allocating functionality is similar to that used in functionality-based design methods, such as most object-oriented design methods, but with a

variation to treat features. Thus, two design guidelines (DG) should be considered to allocate functionality:

DG_1. allocating functionality based on use cases. Applying use cases that pertain to the parent module aids the domain architect to obtain a more detailed understanding of the functionality distribution. Thus, every use case of the parent module must be represented by a sequence of responsibilities within the child modules. Assigning responsibilities to the children during the decomposition leads to the discovery of possible needs for information exchange. This creates a relationship among the modules that needs to be considered. However, it is not important to define how the information is exchanged yet. These questions are dealt with later in the design process.

DG_2. allocating functionality based on features. During the domain analysis, a set of common features and its variability have been identified and represented through the feature model. This information is very important to design a flexible architecture, adaptable for several applications, and thus the process of allocating functionality should consider it. In this way, in the approach, two features F1 and F2 are related if the following conditions hold:

i. if they belong to a same functional category as defined by clients. This functional category may be based on system, module and functional classification;

ii. if they use common data or information;

iii. if there is some strong degree of dependency among the features; and

iv. if they belong to the system layer and they process related system operations or transactions.

These activities should be sufficient to gain confidence that the future applications in the domain will deliver the desired functionality. Although, to check if the requirements (derived from domain analysis) can be met, the approach needs more than just to allocate responsibilities. Thus, after allocating the functionality, the domain architect represents the architecture with views. According to Bass et al. (2003), *a view is a representation of a coherent set of*

architectural elements, as written by and read by system stakeholders. It consists of a representation of a set of elements and the relations among them.

Thus, one view from each of the three major view groups (Bass et al., 2003) (module, concurrency, and deployment) can be used. Figure 6.3 shows the module view of the starship game domain. According to the Figure, the modules *entity* and *event* are dependents of *controller* and all the modules use the services offered by *util* module. After concluding this sub-activity and obtaining the initial domain architecture, the domain architect starts a more detailed process of designing each module with its variability representation.

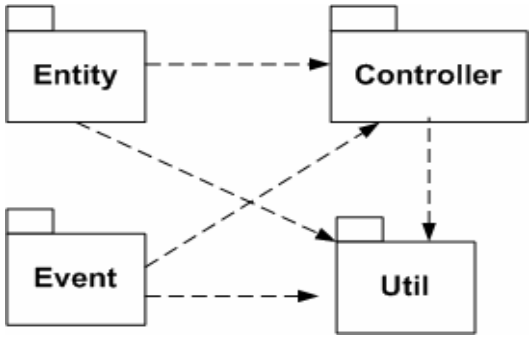

Figure 6.3. Module View of the Starship Game Domain.

6.5. Represent Variability

According to Svahnberg et al. (2001), variability is the ability to change or customize a system. Improving variability in a system implies making it easier to do certain kinds of changes. Moreover, it is possible to anticipate some types of variability and construct a system in such a way that it is prepared for inserting predetermined changes. In DSSA design, the need for variability is even bigger, since flexibility is a crucial requirement. However, even with about thirteen approaches available in the literature (*Aggregation/delegation, Inheritance, Parameterization, Overloading, Delphi properties, Dynamic Class Loading, Static Libraries, Dynamic Link Libraries, Conditional Compilation, Frames, Reflection, Aspect-oriented programming,* and *Design Patterns*) (Anastasopoulos & Gacek, 2001), in general, software architects do not have effective ways to do it (Coplin et al., 1998). In RiDE, Design Patterns (Gamma et al., 1995) are used, but together with useful guidelines that determine how and when patterns can be used to represent the different kinds of variability that can exist in a DSSA. Keepence & Mannion (1999) and Lee & Kang (2004) also use

design patterns. However, their approaches just present some patterns that can be used, without discussing useful guidelines on how and why each pattern should be used for each situation.

Thus, in order to design the variability of each module, we consider that it should be traceable from domain analysis assets (features) to architecture, according to *alternative*, *or* and *optional features* (Lee & Kang, 2004).

Alternative features indicate a set of features, from which *only one* must be present in an application. Thus, the following set of patterns can be used (Gamma et al., 1995):

1. Abstract Factory and Singleton. The *abstract factory* pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes. Thus, it can be used in the following way: one *abstract factory* acts as interface for the remainder of the architecture, while in one application, just one concrete factory will exist for one of the alternative features. The *singleton* pattern ensures that a class has only one instance, and provides a global point of access to it. In this way, this pattern can be used to assure that only one feature can be present in the application.

2. Factory Method. The *factory* pattern defines an interface for creating an object, but lets subclasses decide which class to instantiate. This pattern is similar to the *abstract factory* and can be used also for alternative features.

3. Prototype and Singleton. The *prototype* pattern specifies the kinds of objects to create using a prototypical instance, and creates new objects by copying this prototype. In this pattern, the prototype specifies how the interaction with the feature should be, by defining a concrete prototype for each feature. Thus, depending on which feature will be in the application, just one concrete prototype will be used.

4. Strategy. The *strategy* pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Using this pattern, an abstract strategy represents the interaction with the remainder of the architecture, and only one concrete strategy will be present. It is similar to the *factory method*, but works in the behavior level.

5. Template Method. The *template method* defines the skeleton of an algorithm in an operation, deferring some steps to subclasses. This pattern is close to *strategy*; however, it forces the implementation to follow some pre-defined steps. It is also useful when the interaction between the alternative feature and the remainder of the architecture needs a more intensive communication.

In the case of *Or features*, they represent a set of features, from which *at least one* must be present in an application. For this type of feature, two patterns can be used (Gamma et al., 1995):

1. Builder. The *builder* pattern separates the construction of a complex object from its representation, so that the same construction process can create different implementations. This pattern can be used to build composed features. Thus, for the remainder of the architecture, only the *Director* is available, being responsible to decide which features will be in the application and which will not. For example, the *Director* can be responsible to insert at least one of the features.

2. Observer. The *observer* pattern defines a one-to-many dependency between objects so that when one object changes its state, all of its dependents are notified and updated automatically. Using this pattern, features can be added to the application as a plug-in, after the deployment.

Optional features are features that *may or may not* be present in a application. For this type of feature, three patterns can be used (Gamma et al., 1995):

1. Builder. In *optional features*, this pattern can decide if a feature will be present in an application or not.

2. Decorator. The *decorator* pattern can be used for optional features, mainly those that are additional features. Thus, if a feature is present, the *ConcreteDecorator* is responsible to manage and call the execution. For example, in a simple search system, after returning the results, nothing is done, but, if a feature to learn with the user is present in the application, after returning the result, one *Decorator* can perform the tasks of storing the search words and the obtained results, making a user profile.

3. Observer. This pattern can be used in the same way as in *or features*.

6.5.1. Guidelines for using Patterns

Seeking a more detailed process, the following additional guidelines are defined for each type of feature:

Alternative features. When a feature can be directly mapped into a single class, we suggest the use of the *Prototype* pattern, because it is simpler and allows to instantiate a specific object, depending on the feature that is used in the application, through simple inheritance. Another suggestion is to use the *Singleton* pattern to keep and manage a unique instance of this class.

However, there could be cases where one feature can only be implemented by more than one class. In this case, it is recommended to use the *Abstract Factory* pattern or *Factory Method*, which allows, through inheritance of the factory, to build more than one class (*ConcreteProduct*) for the same feature. The difference between them is that in the *Abstract Factory* there is a separate class for instantiating the objects, while in the *Factory Method* the instantiation of the objects is incorporated into another class of the architecture. The *Factory Method* is simpler, and should be preferably used. However, it impedes inheritance to be used in the class that encapsulates it. Thus, the *Abstract Factory* should be used if inheritance is to be allowed.

In this case, the *Singleton* pattern can also be used, but together with the *Facade* pattern, in a way that all classes that are related to the feature are united behind a single class. In order to facilitate the construction of the features that require more than one class, together with *Abstract Factory* or *Factory Method*, the *Builder* pattern, which facilitates object construction, is recommended.

To map the behavior, we suggest the utilization of the *Strategy* and *Template Method* patterns, which, through simple inheritance, use only one of the subclasses as the implementation for the behavior.

Alternative features were not common in the starship game domain, however, we believe that the discussed guidelines were defined in a reasonable way and can be useful in other application domains.

Or features. In order to assure that at least one feature is present, and to manage and determine which features are present, we suggest the use of the *Builder* pattern, with the *Director* being responsible for managing these aspects. The *Composite* pattern can also be used to manage the features composition structure, providing the application discovery mechanisms, to find out which features are present and which are not.

The *Adapter* and *Bridge* patterns can also be used to separate the features from the remainder of the architecture. Thus, depending on the number of features that are present, a different adapter or bridge may be used. The difference is that the *Adapter* is more generic, and can be used to unite several classes under a single interface, where the signatures of the method implementations can be different from the specification. The *Bridge* is more appropriate when a single class is needed, and the signatures of the methods implementations are identical to the specification.

The *Decorator* and *Chain of Responsibility* patterns may be used when different features have functionalities that are complementary to each other, so that one feature must execute after the other. The *Decorator* is a structural pattern, more indicated for the cases where the interaction between several features is complex, but well defined, because in these cases, the structure of the additional functionalities must be defined during design time. The *Chain of Responsibility* is a behavioral pattern, i.e., the structure of the interaction is not well defined, and thus it is more indicated for the request-response model. If the interaction between the features is simpler, we suggest the use of *Chain of Responsibility*, preferably together with the *Command* pattern to represent the request.

Figure 6.4 show two examples using the defined guidelines. In the first one, the *Builder* pattern is used to manage the composition of each element (*Enemy* and *Main Character*). The second one, the *Chain of Responsibility* pattern is used to identify operations and delegate for specific classes.

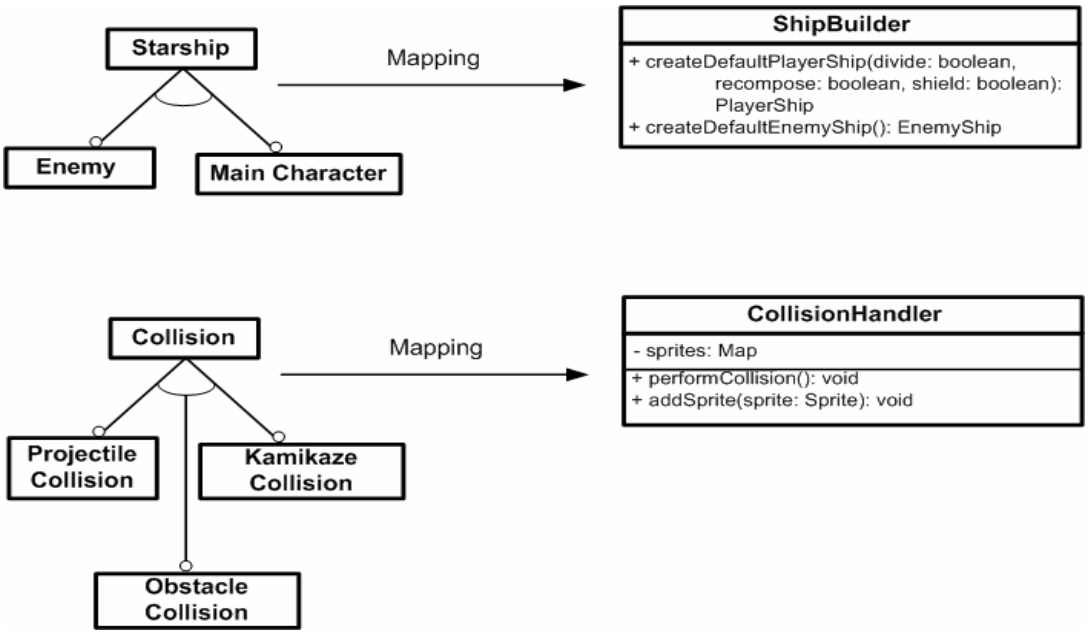

Figure 6.4. Variability Representation in Or Features.

Optional features. For optional features, the same set of patterns used for the *Or features* can be used, with the difference that in this case it is not necessary to guarantee that at least one feature is present in the application.

Figure 6.5 shows an example using the guidelines. In this case, the *Strategy* pattern was used to define which types of actions can be adopted by the starship.

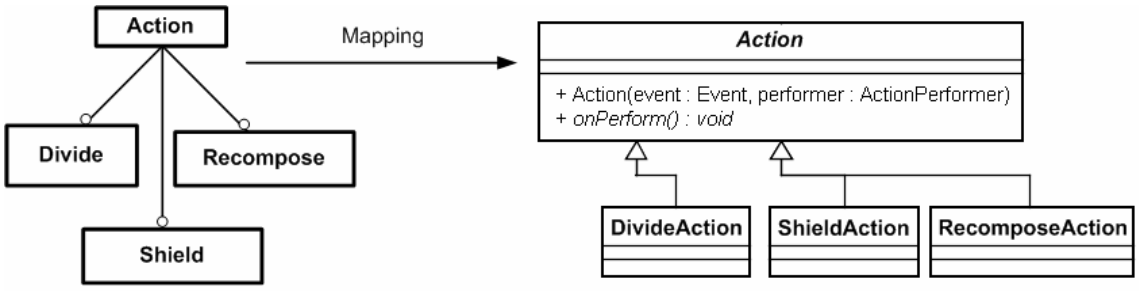

Figure 6.5. Variability Representation in Optional Features.

6.6. Define Component

After decomposing the modules, refining them, and representing their internal variability, the domain architect starts to define the architecture components. In order to do it, three sub-activities are performed, as shows Figure 6.6. There are

relationships among the sub-activities; however, these are not demonstrated in the Figure due to legibility.

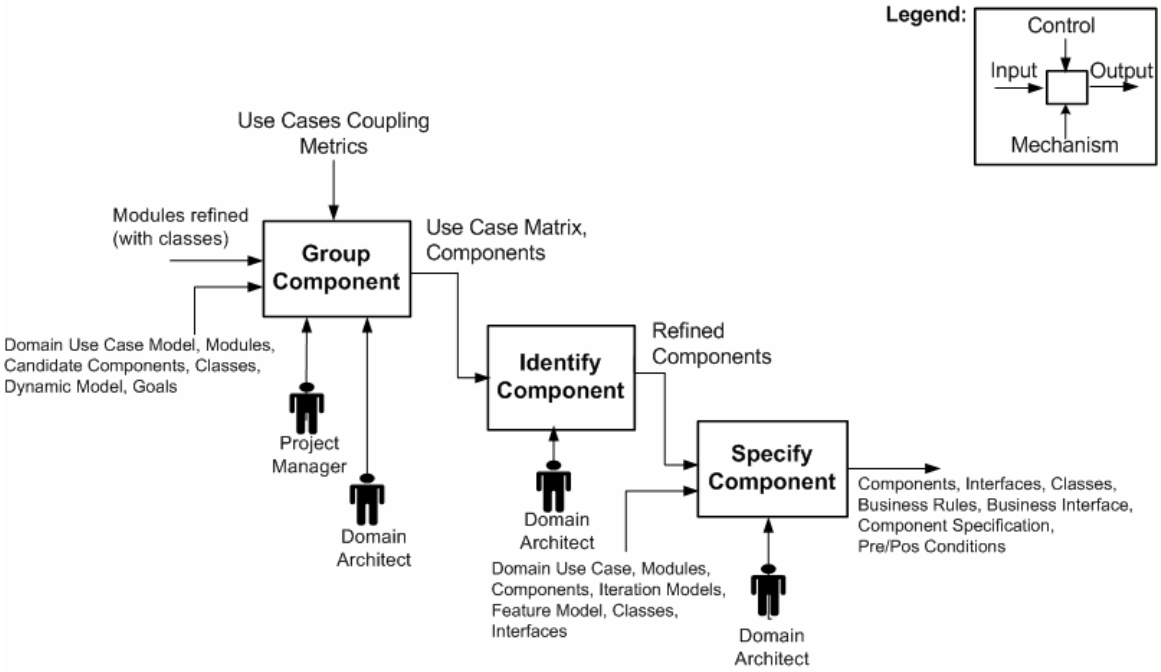

Figure 6.6. Sub-Activities for Defining Components.

6.6.1. Group Component

After performing the module refinement with classes and their variability, the domain architect starts the component grouping sub-activity. This sub-activity is composed by four steps: *Measure Functional Dependency, Cluster Use Cases, Allocate Classes to Components,* and *Select Candidate Components* (Appendix E presents the templates for these steps). There are other works on component identification and grouping (Sugumaran et al., 1999), (Jain & Chalimeda, 2001), (Kim & Chang, 2004), (Blois et al., 2005). However, most of them emphasizes the principle of functional independence and do not provide systematic steps, metrics and guidelines. In this sub-activity, our work is close to (Kim & Chang, 2004) (two first activities), with some improvements that will be described in detail in the next Sections; and (Blois et al., 2005) in the sense of defining steps, metrics, guidelines and roles.

6.6.1.1. Measure Functional Dependency

In this sub-activity, the domain architect, in conjunction with the project manager, assesses the functional dependencies between use cases and cluster closely related use cases into a component. In order to do it, the following Criteria (C) are used to measure the functional dependency between two use cases, UC_i and UC_J:

C_1. Criterion concerning Sub-Systems. If both UC_i and UC_j belong to the same sub-system, the measure for this criterion M_{11} gets the value 1. Otherwise M_{11} gets 0. If the target system is so small that sub-systems are not defined, then M_{11} also gets 1.

C_2. Criterion concerning Actors. Use cases initiated or invoked by the same actor are more closely related than the ones that are not. Thus, measuring this relationship, we should consider that a use case may be initiated by more than one actor. The measure for this criterion, M_{12}, is defined as the following:

$$M_{12} = \frac{(\text{actors initiating } U_i \cap \text{actors initiating } U_j)}{(\text{actors initiating } U_i \cup \text{actors initiating } U_j)}$$

In the metric, M_{12} measures the proportion of actors that initiate both U_i and U_j, against the set of all related actors. A higher value of M_{12} indicates a higher degree of use cases sharing the same actor (s). The range of M_{12} is 0..1. If M_{12} is 1, U_i and U_j have same actor (s), but if M_{12} is 0, then there is no actor initiating both use cases.

C_3. Criterion concerning Shared Data. Use cases manipulating the same set of data are more closely related than other use cases. Thus, the degree of commonality on data manipulated by two use cases M_{13} can be more accurately expressed using a metric rather than simply representing it as a Boolean value or a scale. The measure for this criterion is defined as the following:

$$M_{13} = \frac{(\text{classes manipulated by } U_i \cap \text{classes manipulated by } U_j)}{(\text{classes manipulated by } U_i \cup \text{classes manipulated by } U_j)}$$

The range, minimum value and maximum value of this metric are the same as those for M_{12}.

C_4. Criterion concerning Use Case Relationship. A measure, M_{14}, represents the degree of coupling between two use cases. Three types of relationships among use cases exist: <<*include*>>, <<*extend*>> and *generalization*. If UC_i and UC_J are specialized from a generalized use case, then M_{14} is 1. If UC_i and UC_J are related with <<extend>>, then M_{14} gets 1. If UC_i includes UC_J with <<*include*>>, then the relationships between UC_J and use cases other than UC_i should also be considered. For example, UC_J may also be included by UC_k. In this case, if UC_i has a relatively strong relationship with UC_k, then M_{14} gets a value of 1. If UC_i is independent from UC_k, UC_J should be placed in a separate component.

Each criterion may be assigned with a different weight value, and the sum of all weights must be 1. Based on the given criteria, we use a metric (Kim & Chang, 2004) to compute the functional dependency between two use cases UC_i and UC_J, FD_{ij}, as follows: $FD_{ij} = \sum(M_k * W_k)$, where M_k is the measure on the k^{th} criterion for two use cases, and W_k is the weight value for the k^{th} criterion. The range of FD_{ij} is between 0 and 1. The template presents in Appendix E (Use Cases Clustering) is used to document this sub-activity.

6.6.1.2. Cluster Use Cases

After computing the functional dependencies for each pair of use cases, the domain architect defines candidate components by clustering related use cases. In order to register it, the use cases are organized into a worksheet (Appendix E) where the dependencies are established and calculated as shows Table 6.1.

Table 6.1. Functional Dependency Matrix.

	UC_1	**UC_2**	**UC_3**	**UC_4**	**UC_5**	**UC_6**	**UC_7**	**UC_8**
UC_1	1	0.7	0.6	0.2	0.3	0.4	0.5	0.3
UC_2	0.7	1	0.7	0.3	0.2	0.5	0.4	0.2
UC_3	0.6	0.7	1	0.4	0.2	0.4	0.3	0.4
UC_4	0.2	0.3	0.4	1	0.6	0.8	0.9	0.1
UC_5	0.3	0.2	0.2	0.6	1	0.7	0.8	0.2
UC_6	0.4	0.5	0.4	0.8	0.7	1	0.7	0.1
UC_7	0.5	0.4	0.3	0.9	0.8	0.7	1	0.2
UC_8	0.3	0.2	0.4	0.1	0.2	0.1	0.2	1
.....								

The clustering algorithm used for this task uses a row and column shifting method, and requires a constant value *t* to identify rows and columns to be shifted. For a given value of *t*, the algorithm performs the best possible clustering. As *t* increases, smaller grained components are acquired. However, the value of *t* is chosen from values of functional dependency computed by the domain expert. Different values of *t* can be applied in this step, and different clustering results can be examined by domain experts or developers to select the optimal result.

After calculating the functional dependency and shifting each use case, the next task is to identify the set of adjacent use cases that have FD_{ij} values greater than *t*. Thus, each set of clustered use cases makes up a candidate component. Table 6.1 shows an example of this sub-activity. In this case, for example, if the domain architect defines a *t* value equals 0.6, can be obtained two candidate components: the first, including the use cases UC_1, UC_2, UC_3 and the second composed of UC_4, UC_5, UC_6, UC_7.

A possible problem during clustering occurs when we can not have a clear-cut clustering. A use case can be shifted several times if the use case has more than one functional dependency FD_{ij} that is greater than *t*. This may yield a clustering result with use cases shared by sets (S), and as the result, clear-cut clustering is not possible. This situation is showed in Figure 6.7 (Kim & Chang, 2004).

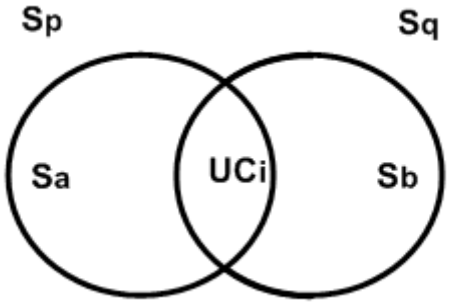

Figure 6.7. Relationship among Sets of Use Cases.

In the Figure, S_p and S_q are clustered sets of use cases. S_a and S_b are sets of use cases included exclusively into S_p and S_q, respectively. A use case UC_i is the intersection of S_a and S_b. That is, S_p is union of S_a and $\{UC_i\}$, and S_q is also a union of S_b and $\{UC_i\}$.

We may focus on the relationship between S_a, S_b and UC_i to derive a difference as follows; a degree of coupling between a set and a use case as *Coupling* (S_a, UC_i); If UC_j is an element of S_a, *Coupling* (S_a, UC_i) is $\sum_{FDij}$ for all i. We calculate the difference between (*Coupling* (S_a, UC_i) and *Coupling* (S_b, UC_i)). If the difference is greater than o, UC_i should be included in S_p. If the difference is less than o, UC_i should be included in S_q. If the difference is equal to o, then the use case may be arbitrarily allocated, or allocated by domain experts.

6.6.1.3. Allocate Classes to Components

After performing the clustering process, the domain architect starts to allocate classes to each component using the dynamic model expressed by sequence diagrams. In our approach, a component includes several use cases. The dynamic behavior of each use case is depicted by a sequence diagram, and a sequence diagram specifies a set of participating objects/classes as shows Figure 6.8.

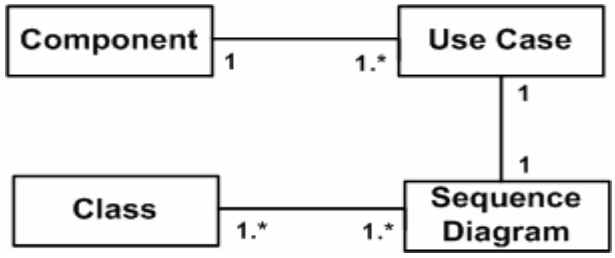

Figure 6.8. Relationship among the assets.

Thus, for each component identified in the previous steps, the domain architect locates sequence diagrams for use cases included in the component. Next, the participating classes that are shown in a sequence diagram are assigned to the corresponding component. In order to register it, a worksheet (Appendix E) matrix can be used, relating classes and their associated components. However, sometimes the assignment of classes into components may yield a conflict where a class is assigned into more than one component.

For the conflicts, we consider two cases: the *first one* is the dynamic model. In this case, we first compute an entity dependency between a component and a class, and then compare the entity dependency between one

conflicting class from one component and one from another component. In order to compute the entity dependency, we used the following criteria:

i. Coupling and cohesion. Initially, the coupling and cohesion can be computed for the classes in order to solve the conflicts.

ii. The number of messages. After analyzing coupling and cohesion, the number of messages being passed can be considered. Through tracing a sequence diagram, we can estimate the number of dynamic messages that are exchanged between classes for each use case during runtime. This criterion, M_{21}, is computed as following:

$$M_{21} = \sum \textit{the number of messages passed to } C_i \textit{, for all use cases in } Com_j$$

iii. Relationship between classes. In general, there are fives types of relationship among two classes: *inheritance*, *association*, *dependency*, *aggregation* and *composition*. Instead of defining constant values for the weights, we define the following comparison on different relationships: 0 < *Dependency* < *Aggregation* < *Association* < *Inheritance* < *Composition* < 1. Using this weight values, we can compute the inter-class relationship between a pair of classes, defining a constant value *t*, to determine whether two classes should be put together in a component or not:

$$M_{22} = \sum \textit{weight of relationship between } C_i \textit{ and other classes in } Com_j$$

From the criteria, we use a formula (Jain & Chalimeda, 2001) to compute the entity dependency $ED_{ij} = (M_k * W_k)$ between a class C_i and a component Com_j.

The *second case* to treat conflict is based on static relationship. In this case, the strength of the static relationship (S_{ij}) is defined as:

$S_{ij} = W_s \text{ x } N_{ij}$, where W_s = the static association weight (0 < Dependency < Aggregation < Association < Inheritance < Composition < 1), N_{ij} = the number of associations between class *i* and *j*. The S_{ij} range is scaled on a 0 to 1 scale. Thus, based on static relationship, the domain architect can decide which classes will be allocated to each component.

6.6.1.4. Select Candidate Components

Following the previous steps, the domain architect, in conjunction with the project manager, identifies candidate components. The value of *t* used in the process defines the number of components and their granularity. Thus, it is recommended to apply different values of *t* to generate different clustering results and to let architects and project managers choose an optimal clustering result using the criteria. In this sense, the goal of this step is to choose an optimal configuration from different clustering results generated by applying different *t* values.

As previously defined, the range of *t* is between the minimum and maximum values of all measured functional dependencies, i.e. from 0.1 to 0.9. Thus, a smaller value of *t* produces components with large granularity; on the other hand, the total number of components is decreased. In contrast, a larger value of *t* produces finer grained components and the total number of components is increased.

Additionally, two other criteria can be used to select the candidate components: *costs* – the project manager can decide which components will be specified according to a predefined cost (man/hour) – and *complexity* – as the components are based on use cases, and the use cases have a effort metric estimated, the project manager can select components according to their *complexity* (low, medium, high).

6.6.2. Identify Component

Once the component grouping sub-activity is finished, the domain architect refines the identified components. This refinement consists in analyzing the set of components according to the initial specifications (feature models, domain use case model) in order to assure that the identified components match their specifications.

6.6.3. Specify Component

In this sub-activity, the goal is to create an initial set of interfaces and component specifications. This sub-activity is composed of three steps: *Identify Interfaces*, *Identify Core Classes*, and *Refine the Specification*.

6.6.3.1. Identify Interfaces

In our approach, we consider two types of interfaces: *system* and *business*. An interface is a set of operations, with each operation defining some services or function that the component will perform for the client (Szyperski, 2002). The system interfaces and their operations emerge from a consideration of the feature model and mainly of the use case model. This interface is focused on, and derived from, system interactions.

Thus, in order to identify system interfaces for the components, the domain architect uses the following approach: for each use case, he considers whether or not there are system responsibilities that must be modeled. If so, they are represented as one or more operations of the interfaces (just signatures). This gives an initial set of interfaces and operations.

The business interfaces are abstractions of the information that must be managed by components. Our process for identifying them is the following: *to analyze the feature model to identify classes* (*for each module and component*); *to represent the classes based on features with attributes and multiplicity*; and *to refine the business rules using formal language*. Sometimes, depending of the process that was performed by the domain architect in the previous steps, the process to identify business interfaces becomes just a refinement step.

Additionally, for each operation defined in the interfaces, the *pre-* and *post-* conditions can be specified using a formal language such as Object Constraint Language (OCL) (Warmer & Kleppe, 1999).

Figure 6.9 shows the *Audio* component, using the UML 2.0 notation (OMG, 2005), responsible to trigger sound events according to game's actions and its business interface to play music and specific sounds such as new score, game over, etc.

Figure 6.9. Audio Component.

6.6.3.2. Identify Core Classes and Refine the Specification

After identifying the interfaces, the domain architect decides which classes from each module are in the core (Cheesman & Daniels, 2000). A core class is a business type that has independent existence within the business, characterized by the following: a *business identifier*, usually independent of other identifiers; and *independent existence*, with no mandatory associations with other types.

The purpose of identifying core classes is to start defining which information is dependent on others, and which information can stand alone. It is useful to allocate information responsibilities to interfaces and to specify precise components. According to 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*". The general rule is that we create one business interface for each core class; thus, each business interface manages the information represented by the core class.

At the end, for every component that is specified, the domain architect defines which interfaces its realizations must support (*provided* and *required* interfaces), and documents the Component Specification Template (CST) (Appendix F presents more details about this template). The CST is a template that comprehends the component specification, consisting of: *component name, brief description, workflow for each use case, contained classes, use cases and requirements referred, provided and required interfaces, and commonality* and *variability represented by features*.

6.7. Represent Domain Architecture

Once the component specification is performed, the domain architect represents the initial domain architecture based on components. In order to do it, architectural views and component diagrams are used to show the components, their interconnection, and the provided and required interfaces (Appendix G presents the template for this activity).

Figure 6.10 shows the domain architecture of the starship game domain with its components, interfaces and relationships among them. The interface specifications were omitted in order to improve the visibility of the figure. The main goal of the component *Strategy* is to manage the different strategies of movement of each enemy. *Screen* is responsible to manage the transition of each screen of the game. *Action*'s goal is to define the actions of each event. *Movement* main function is to execute the kinds of movements performed by players and enemies. At the end, the components *Audio*, *Entity*, *Event*, and *Graphic* are responsible to manage the sounds, entities, events, and graphics used in each game of the domain. The *Game Engine* works as a controller for the games. It is responsible to manage the requests and calls for the others components, since the games need to have an engine to keep the idea of continuity.

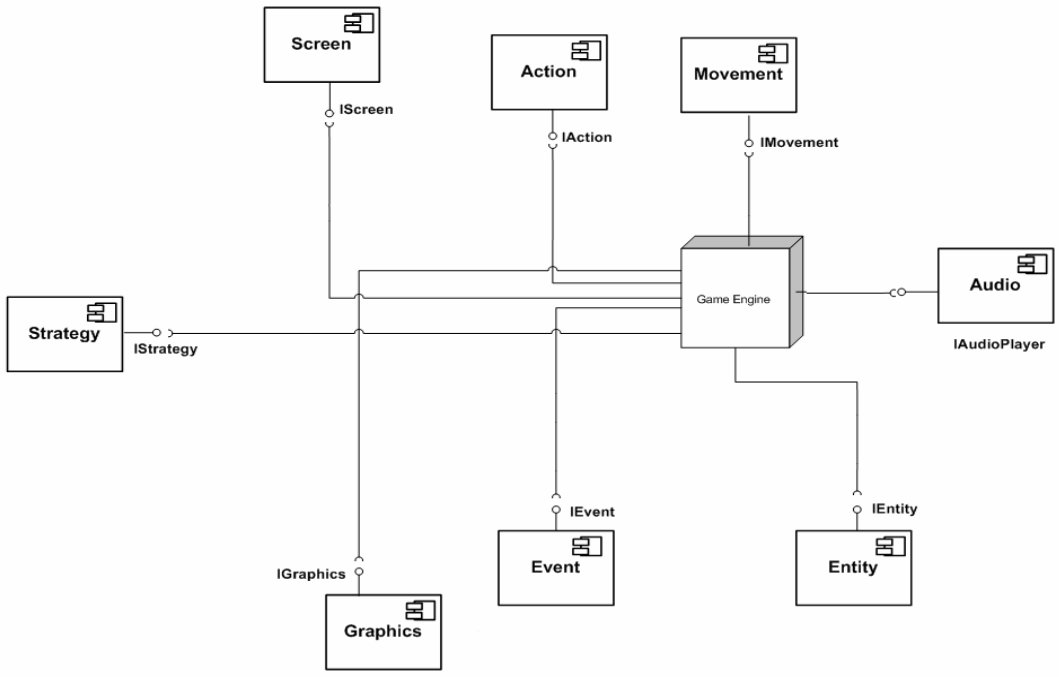

Figure 6.10. Domain Architecture of the Starship Game Domain.

During this activity, the domain architect can discover and refine other components, using, for example, collaboration diagrams.

6.8. Summary

Table 6.2 presents a summary of the approach, with Activities (A), Sub-Activities (S), Inputs (I), Outputs (O), and the Roles (R).

Table 6.2. Summary of the Domain Design Step.

A	S	I	O	R
Decompose Module	-	Domain Use Case Model, Feature Model	Modules	Domain Architect, Domain Analyst, Project Manager
Refine Module	Choose the Architectural Drivers	Feature Model, Scenarios	Modules refined, initial architecture (with classes)	Domain analyst, Domain Architect
	Choose the Architectural Patterns	Modules, Domain Use Case Model, Feature Model, Constraints, Scenarios		
	Allocate Functionality using Views	Domain Use Case Model, Feature Model		
Represent Variability	-	Feature Model, Domain Use Case Model	Modules refined (with classes)	Domain architect
Define Component	Group Component	Domain Use Case Model, Modules, Candidate Components, Classes, Dynamic Model, Goals	Use Case Matrix, Components	Project Manager, Domain Architect
	Identify Component	Use Case Matrix, Components	Refined Components	Domain Architect
	Specify Component	Refined Components, Domain Use Case Model, Modules, Components, Iteration Models, Feature Model, Classes, Interfaces	Components, Interfaces, Classes, Business Rules, Business Interface, Component Specification, Pre/Pos Conditions	Domain Architect
Represent Domain Architecture	-	Components, Interfaces, Classes, Business Rules, Business Interface, Component Specification, Pre/Pos Conditions	Component Diagrams, Domain Architecture	Domain Architect

6.9. Other Directions in Domain Design

The literature does not present many works related to domain design. However, different directions related to software architecture, software reuse processes, and component-based development can be highlighted.

Due to the importance of architecture in the software development process, researchers and practitioners have proposed some works related to software architecture design methods, such as ADD (Bass et al., 2003), CBAM (Bass et al., 2003), DAGAR (STARS, 1996) and functionality-based design (Bosch, 2000). However, except for the third one, these methods are more appropriated to single systems development, where focus is not explicit on reuse (Matinlassi, 2004). DAGAR, although defined to work with reuse, has some drawbacks, such as to be directed to Ada language, and too generic to be used nowadays. The DSSA work proposed by Tracz (1995) was used as inspiration in the approach of this thesis; however, the approach presented here has more details, guidelines, tasks, and roles in order to systematize the process.

In a certain sense, the software reuse processes (discussed in Chapter 3) served as an initial inspiration to the presented approach, mainly, in its weak and strong points.

Component-Based Development methods, such as Catalysis (D'Souza & Wills, 1998) and UML Components (Cheesman & Daniels, 2000), although relevant in CBD context, also present gaps when the focus is on mass customization software. In Catalysis, for example, the flows among tasks are not completely defined and concrete guidelines for component identification are not provided. Additionally, the method does not deal systematically with variability and domain architecture. The same problem with domain architecture is seen in UML Components.

Other relevant approaches, such as (Lee et al., 1999), (Chen, 2002), and (Choi et al., 2004) are also not focused in developing DSSA and do not consider issues such as variability and component identification and grouping.

6.10. Chapter Summary

Domain design is a critical aspect in a domain engineering process since its main goal is to develop the domain-specific software architecture for a set of applications in a domain. However, the existing approaches do not present a systematic way of how to do it, since they are not completely close to domain design. Some are related to software architecture in general and other to component-based development. Moreover, the software reuse processes (as presented in Chapter 3) have different emphasis on steps of domain analysis, design, and implementation, not covering all their activities in details.

In this context, this Chapter discussed the second step of RiDE, the domain design, based on a set of guidelines, metrics, inputs, outputs, activities, sub-activities, and roles. The next Chapter will present the last step of the process: the domain implementation, where the main goal is to implement reusable assets based on the assets previously defined in domain analysis and domain design.

The Domain Implementation Step

"Components will exist only where component vendors and component clients join forces in sufficient numbers to reach a critical mass"

Clemens Szyperski(2002)
Software Component Specialist

The goals of the domain implementation step are to provide the detailed design, implementation, and documentation of reusable software assets, based on the domain-specific software architecture. The reusable software assets are mainly reusable components and interfaces; however, other artifacts, such as documentation are also products of domain implementation. Additionally, domain implementation incorporates configuration mechanisms that enable application engineering to select variants and build an application reusing components and interfaces.

In this context, this Chapter presents the domain implementation approach which is the third and last step of the domain engineering process, its principles, activities, sub-activities, and roles. The approach is based on the strong and weak points of the reuse processes discussed in Chapter 3 (focusing on the implementation step) in conjunction with other relevant directions in the component-based development area.

7.1. Introduction

In the early days of computing, software started with simple programs by implementing algorithms (Dahl, et al., 1972). Nevertheless, the problem domain that could be supported with software constantly grew. The systems to be built

also became constantly more complex, and the teams to work on a single software system continued growing (Sametinger, 1997). In this sense, since 1968[1] (McIlroy), Component-Based Development (CBD) is seen by many software engineers and researchers as a promising approach to reduce costs and increase the software quality. In the industrial world, it can be seen with several case studies published (Crnlovic & Larsson, 2000), (Sparling, 2000), (Baster et al., 2001), (Kim, 2002). On the other hand, in the academic world, this issue can be identified through books (Sametinger, 1997), (Heineman & Councill, 2001), (Szyperski, 2002), special issues (Meyer & Mingins, 1999), and specific conferences around the world[2]. CBD can be understood as a set of techniques, tools, and processes for designing and building applications reusing specific assets (Greenfield et al., 2004).

The reuse of software components is important in a variety of aspects for software development (Brown & Wallnau, 1998). As many software systems contain many similar or even identical components (Weiss & Lai, 1999) that are developed from scratch has led to efforts to reuse existing components (Sametinger, 1997). Structuring a system into largely independent components has several advantages. It is easier to distribute the component among various engineers to allow parallel development (Repenning et al., 2001). Moreover, maintenance is easier when clean interfaces have been designed for the components, because changes can be made locally, without having unknown effects on the whole system. At the end, if the component interrelations are clearly documented and kept to a minimum, it becomes easier to exchange components and incorporate new ones into a system.

On the other hand, for the existence of software components, software component technology needs to meet with proper software component markets. The market cannot exist without, possibly primitive, technology and, equally, the technology cannot be sustained or evolved without a sufficiently strong market (Szyperski, 2002). Analyzing this aspect, is possible to identify that the

[1] In this period, the term software crisis was coined as presented in Chapter 2.

[2] Just in 2006, there were six conferences on the theme: i. *Asian-Pacific Software Engineering Conference* (APSEC) whose main topic was CBD, ii. *Brazilian Workshop on Component-Based Development*, iii. *EUROMICRO Software Engineering Conference*, iv. *Generative Programming and Component Engineering* (GPCE), v. *International Symposium on Component-Based Software Engineering* (CBSE), vi. *Workshop on Component-Oriented Programming* (WCOP) in conjunction with *European Conference on Object-Oriented Programming* (ECOOP).

software component concept is close to domain engineering, since the components are developed in a specific domain, increasing the reuse potential. Moreover, according to Peter Freeman (1987) – one of the pioneers in software reuse – the reuse of source components implicitly involves the reuse of analysis and design specifications, motivating even more the integration among domain analysis, domain design, and domain implementation, in which the key goal is to provide the implementation and documentation of reusable assets (components).

The literature has several contributions for the Domain Analysis (Section 5.7.) and Domain Design (Section 6.9.) fields, but the Domain Implementation area presents some gaps in reuse processes and specific methods and approaches for component-based development that need to be closed to allow its use in large-scale. The main problem is that the current published research does not define how to implement and document software components in a systematic way.

In this context, the next Sections present the third step of the domain engineering process: domain implementation; based on a coherent set of principles, roles, inputs and outputs.

7.2. Initial Considerations

As stated, the key goal of this step is to implement and document software components. In this way, the component concept should be strongly understood. Thus, Szyperski's (2002) definition is used as basis (pp. 41): "*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***".

In order to do it, the proposed approach is based on a set of principles that help the software engineer during the development of software components as described by Szyperski's definition. The approach and its principles are generic, and can be used with different implementation technologies. In RiDE, however, in order to provide a more concrete example, will be used the OSGi

technology to manage the interaction and lifecycle of the implemented components. Thus, next Section briefly describes this technology.

7.2.1. OSGi

Open Service Gateway Interface (OSGi) is a Java-based interface specification that defines a standardized, component-oriented, computing environment for networked services. The scope of the OSGi platform is (OSGi, 2005): a standard, *non-proprietary*, software component framework for manufacturers, service providers, and developers; a powerful model for co-existence of different components/applications in a single JVM; a flexible deployment API that controls the life cycle of applications; a secure environment that executes applications in a sandbox; and a cooperative model where applications can dynamically discover and use services provided by other applications running inside the same OSGi platform among others.

A key principle of OSGi is asynchronous management, which means that running applications must be prepared to react immediately to management interactions. For instance, a software component may be upgraded and applications using the component must be able to migrate easily from the old component to the new one. This feature makes OSGi a dynamic environment.

In the current Release (R4), the OSGi software has the following layers (OSGi, 2005):

- **Security Layer**: this layer specifies how to sign OSGi assets. This layer builds heavily on Java JAR signing and the *bundle* concept (Paller, 2006);
- **Module Layer**: this layer is responsible for the management of the unit of modularization, called *bundle*. A bundle is comprised of java classes and other resources, which together can provide functions to end users. Once a bundle is started, its functionality is provided and services are exposed to other bundles installed in the OSGi Service Platform;
- **Lifecycle Layer**: the lifecycle layer provides an API to control the security and lifecycle operations of bundles. Bundles can be

started and stopped, clearly demarcating the operational and maintenance mode of the bundles. When a bundle is taken into use, it is often started. This shows that the bundle can start exposing its services or launch processing tasks associated with the bundle. However, before management of the bundle begins, the bundle is stopped. At this point, the bundle should stop exposing its services and stop any processing associated with the bundle (Paller, 2006); and

- **Service Layer**: services are key abstractions in OSGi. The service model is a publish-find-and-bind model. A service is a normal Java object that is registered under one or more Java interfaces with the service registry. Bundles can register services, search for them, or receive notifications when their registration state changes (OSGi, 2005).

7.3. The Principles

In order to define a practical and effective way to perform the domain implementation, the approach is based on a set of Implementation Principles (IP).

IP₁. A component must have interfaces: the functionality of a component is defined by its interfaces. In the approach, a component has one or more provided and required interfaces. Syntactically, interfaces are similar to object class descriptions: they contain attributes and operations. Semantically, provided interfaces specify those services or capabilities that a component offers to other components in order to carry out its own responsibilities. The concept of required interfaces is essential to enabling software plug-and-play (Bronsard et al., 1997). They expose operation invocations that a component makes to other components. On the other hand, components can be connected together to form compositions. A connection is realized by binding a required interface of one component to a provided interface of another one.

IP₂. A component should have a transparent life cycle mechanism: the components need an independent mechanism to manage their life cycle transparently. For example, the *Life Cycle Layer* in OSGi allows bundles to be

managed in one of the following states (OSGi, 2005): *Installed* (the bundle has been successfully installed), *Resolved* (all classes that the bundle needs are available. This state indicates that the bundle is either ready to be started or has stopped.), *Starting* (the bundle is being started), *Active* (the bundle has successfully started and is running), *Stopping* (the bundle is being stopped), and *Uninstalled* (the bundle has been uninstalled, thus, it cannot move to another state). Other technologies for managing the components life cycle, such as Enterprise Java Beans (EJB) (Sarang et al., 2001), for example, can also help in implementing this principle.

IP_3. A component should be configurable: every component, if applicable, should have its own configurable parameters, which are not required to change often. The value of these parameters should come from a configuration file, thus the value of that parameter can be changed without modifications in the source code.

IP_4. A component should have a proper third-party integration mechanism: if a component depends on third-party integration to obtain a service, the third-party API should not be directly used in the component implementation (Ghosh, 2006). One interesting strategy can be to develop abstract classes such as adapters, and to isolate third-party calls inside these adapters. This not only facilitates third-party integration, but also makes the incorporation of different third-party implementations easier to perform. Another advantage of this approach is that if changes occur, the adapter needs to be changed, but the component does not. If there are several adapters, the principle P_3 (*A component should be configurable*) can be useful to manage their configuration.

IP_5. Context independence: software components should state the assumptions made about their context. Thus, components do not need to know which other specific components will they interact with or will be "hard-wired" with them. Components should be freely connected at assembly time, as long as connections have been verified (Bronsard et al., 1997), (Szyperski, 2002).

IP_6. Documentation: component documentation is an important issue because it provides information that assists reusers to assess whether the

component fulfils their requirements and architecture and restrictions for using it.

IP_7. Evolvability: a key premise of plug-and-play is that systems will evolve easily (Bronsard et al., 1997). Thus, components should be added, removed, or replaced after deployment, in a transparent way. The use of a *Life Cycle Layer*, such as the one existent in OSGi, allows these operations to be managed in a coordinated way through command lines or using an administrative visual interface.

IP_8. Version Compatibility: versions of components can change during the evolution of applications. Usually, vendors provide compatibility with previous versions. Nevertheless, the management of version numbers for applications becomes more complex because of the availability of multiple component versions. Also, in some cases a new component may change its API (e.g. methods move up or down the inheritance hierarchy, or signatures change). To ensure compatibility, versions can be checked at bind-time if direct interfaces are used. Immutable interfaces could solve this problem. OSGi solves this problem with its built-in version management mechanisms, which allow bundles written for the new release to adapt to the old implementation.

The approach for domain implementation consists of two activities (Almeida et al., 2007b): **component implementation** and **component documentation**.

Since the following examples are based on OSGi, will be used the term bundle to refer to a software component. However, it is important to stress that this concept of component is the same as known by the component-based development community, and that other implementation technologies can be used with the method as well.

7.4. Component Implementation

In this activity, the software engineer, based on requirements, implements the software components through a set of well defined sub-activities. The approach is intended to be used in the scope of domain engineering, and therefore it depends on assets developed in domain analysis (feature model, requirements,

domain use case model) and domain design (domain-specific software architecture, component specifications).

This activity is divided into two sets of sub-activities, each one with a different purpose. Sub-activities 1 to 4 deal with the *provided* services, i.e. when the software engineer wants to implement a component to be reused. Sub-activities 5 to 7 deal with *required* services, i.e. when the software engineer wants to reuse services from existent components.

Sub-Activity 1. The first sub-activity is to describe the component, providing general-purpose information, such as the component vendor, version, package, among others. This information may be used to identify a component, an important issue when components are stored in a repository, for example. It may also be used to dynamically publish the provided interfaces to be used by other components. In OSGi, this description is performed in the *manifest file*. This file provides information about the component that the OSGi framework needs to correctly install and activate a component. In order to do so, a framework implementation must: process the main Section of the manifest, ignore unrecognized manifest headers, and ignore unknown attributes and directives.

In OSGi's *manifest file*, the following headers should be specified:

- **`Bundle-Activator`**. It specifies the name of the class used to start and stop the component.
- **`Bundle-Category`**. It presents a list of category names for a component, for example, utility, sound. It can be useful to classify the components in a repository, for example.
- **`Bundle-ContactAddress`**. It provides the contact address of component's owner.
- **`Bundle-Copyright`**. It contains the copyright specification for the component.
- **`Bundle-Description`**. It defines a short description of the component.

- **Bundle-ManifestVerstion**. It determines whether the component follows the rules of the specification. It is 1 (the default) for Release 3 components, 2 for Release 4 and later.
- **Bundle-Name**. It defines a readable name for the component.
- **Bundle-SymbolicName**. It specifies a unique, non-localizable name for the component.
- **Bundle-Vendor**. It contains a description of the component vendor.
- **Bundle-Version**. It specifies the version of the component.
- **Export-Package**. It contains a declaration of exported packages.
- **Import-Package**. It declares the imported packages for the component.

In this file, the most important properties are `Bundle-Activator` and `Import-Package`. `Bundle-Activator` informs the framework which class is the `Activator`, which is a kind of main class for the component. On the other hand, the `Export-Package` tells the framework which services can be reused by other components.

Sub-Activity 2. In this second sub-activity, the software engineer should *specify the interfaces*. However, as mentioned before, the domain implementation method depends on artifacts developed in domain analysis and design, such as the domain-specific software architecture and component specifications. These artifacts already contain the interface specification, and so the software engineer only needs to review and refine them, if necessary.

Sub-Activity 3. In the third sub-activity, the goal is to *implement the services* defined in the previous sub-activity, using any implementation technology, as well as the code to *register these services to be used by other components*, if a dynamic execution environment is used. In OSGi, components must have an `Activator` class specified in their manifest file, created in Sub-Activity 1. This class implements the `BundleActivator` interface, which

requires the implementation of two methods, `start` (`BundleContext`) and `stop` (`BundleContext`), used by the software engineer to register the component as listener and start any necessary threads. The `start` method can allocate resources that a component needs, start threads, register services, and more. The `stop` method must clean up and stop any running threads. The relationship between the framework and its installed components is realized by the use of `BundleContext` objects. A `BundleContext` object represents the execution context of a single component within the OSGi platform, and acts as a proxy to the underlying framework. This object is created by the framework when a component is started. The component can use this private `BundleContext` object for the following purposes: installing new component into the OSGi environment, interrogating other component installed in the OSGi environment, obtaining a persistent storage area, retrieving service objects of registered services, registering services in the framework service and subscribing or unsubscribing to events broadcast by the framework.

Sub-Activity 4. In this sub-activity, which concludes the provided side of the component, the goal is to *build and install the component*. According to the implementation technology used, this involves compiling and packaging the component in a form that is suitable to be deployed in the production environment. In OSGi, the software engineer compiles all the files and generates a `jar` file for the component. Next, the component can be deployed in the framework using command lines or a visual interface, such as the Knopflerfish[3] Open Source OSGI Framework, which has an interesting visual administration interface.

Sub-activities 1 to 4 deal with the *provided* side of a component. In order to implement the *required* side, three sub-activities should be performed:

Sub-Activity 5. First, the software engineer needs to describe the component that will reuse other services. This is similar to Sub-activity 1, but with the focus on the services that are required. In OSGi, this is also performed in a *manifest file*, but instead of declaring an `Export-Package` header, an

[3] http://www.knopflerfish.org/

`Import-Package` header must be declared, specifying the name of the service that will be reused.

Sub-activity 6. In this sub-activity, the code that accesses the required services is implemented. Here, different techniques can be employed, such as the use of adapters, wrappers, or other ways to implement this access. The main goal of this sub-activity is to provide low coupling between the required service and the rest of the code, so that it can be more easily modified or replaced. The OSGi specification offers a `ServiceTracker` class that can help in achieving this goal, by tracking the registration, modification, and unregistration of services in an efficient way. In practice, the fundamental tasks of a `ServiceTracker` object are: i. to create an initial list of services as specified by its creator; ii. to listen to `ServiceEvent` instances so that services of interest to the owner are properly tracked; and iii. to allow the owner to customize the tracking process through programmatic selection of the services to be tracked, as well as to act when a service is added or removed (OSGi, 2005).

The `ServiceTracker` interface defines three constructors to create `ServiceTracker` objects, each providing different search criteria (OSGi, 2005):

- `ServiceTracker (BundleContext, String, ServiceTrackerCustomizer)`: this constructor takes a service interface name as the search criterion. The `ServiceTracker` object must then track all services that are registered under the specified service interface name;
- `ServiceTracker (BundleContext, Filter, ServiceTrackerCustomizer)`: this constructor uses a `Filter` object to specify the services to be tracked. The `ServiceTracker` must then track all services that match the specified filter; and
- `ServiceTracker (BundleContext, ServiceReference, ServiceTrackerCustomizer)`: this constructor takes a `ServiceReference` object as the search criterion. The

`ServiceTracker` must then track only the service that corresponds to the specified `ServiceReference`. Using this constructor, no more than one service must ever be tracked, because a `ServiceReference` refers to a specific service.

Each of the `ServiceTracker` constructors takes a `BundleContext` object as a parameter. This `BundleContext` object must be used by a `ServiceTracker` object to track, get, and unget services. Once a `ServiceTracker` object is ready to use, it can track services. The behavior of the `ServiceTracker` class can be customized either by providing a `ServiceTrackerCustomizer` object implementing the desired behavior when the `ServiceTracker` object is constructed, or by sub-classing the `ServiceTracker` class and overriding the `ServiceTrackerCustomizer` methods.

The `ServiceTrackerCustomizer` interface defines the following methods:

- `addingService (ServiceReference)`: called whenever a service is being added to the `ServiceTracker` object;
- `modifiedService (ServiceReference, Object)`: called whenever a tracked service is modified; and
- `removeService (ServiceReference, Object)`: called whenever a tracked service is removed from the `ServiceTracker` object.

When a service is being added to the `ServiceTracker` object or when a tracked service is modified or removed from the `ServiceTracker` object, it must call `addingService`, `modifiedService`, or `removeService`, respectively, on the `ServiceTrackerCustomizer` object (if specified when the `ServiceTracker` object was created); otherwise it must call these methods on itself.

In summary, in this sub-activity the software engineer should *implement the connection between the required services with the rest of the code*.

Sub-Activity 7. The last sub-activity corresponds to *building and installing the component* that reuses the services, which is similar to Sub-Activity 4.

Although these two sets of sub-activities (1-4 and 5-7) are focused on different aspects, in practice they will be present in most components, since normally each component has *both* provided and required interfaces.

Figure 7.1 shows the meta-model for the components developed using the method for domain implementation. According to this meta-model, a component provides the implementation of a set of named interfaces (provided and required); each interface has a set of defined operations; the operations have its input and output parameters; and a syntactic specification associates a type with each of these.

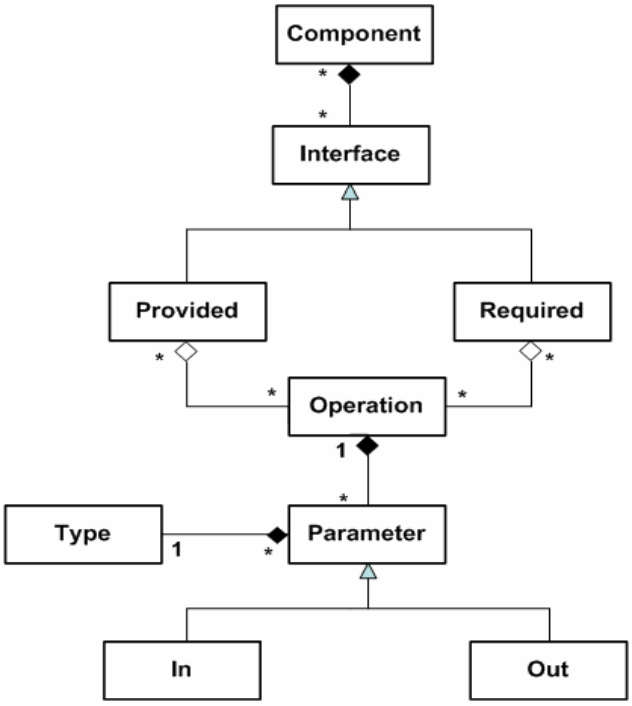

Figure 7.1. Component meta-model.

Once the component implementation is accomplished, the next activity of the approach corresponds to the documentation of the software components.

7.5. Component Documentation

Software documentation can reduce the time and effort to develop new software, increase the ease of porting software to different platforms, and help users to understand software more easily (Phoha, 1997). However, there are problems, such as presented by (Voth, 2004): there is no universally recognized standard for software documentation, in part because documentation style and content differ among programmers and sometimes differ for the same programmer under different circumstances. Additionally, the choice of programming language and the nature of a program may dictate a particular style of documentation that might not easily apply to another environment.

While there is no universally recognized standard for software documentation, there are some directions that can be useful (Phoha, 1997). The first one is a standard for documenting engineering and scientific software, developed by the American National Standards Institute and the American Nuclear Society in 1995; it is called *ANSI/ANS 10-3-1995 Standard for Documentation of Computer Software*. The standard is not a rigid set of specifications, but a guide that can apply to most software projects intended for internal or external use.

The second one is a consortium of software development and component management tool organizations, including IBM Rational, Flashline, LogicLibrary, Borland, and ComponentSource, that defined common conventions for packaging software assets that makes it easier to manage and reuse them in a standard called Reusable Asset Specifications (RAS) (Voth, 2004). However, even with these directions, it is necessary more data and experience reports in order to use it in large-scale.

On the other hand, some research works showed that software documentation is, in practice, poor, incomplete, and too difficult to maintain under industry time pressure (Briand, 2003), (Lethbridge et al., 2003). Additionally, software engineers typically do not update documentation either timely or completely (Lethbridge et al., 2003), however, out-of-date software documentation remains useful in many circumstances (Briand, 2003), (Lethbridge et al., 2003).

In the context of software reuse and component-based development, software documentation is very important, since reusers need to identify, understand, adapt and integrate software components in their applications. Moreover, software components are subject to composition by third parties that sometimes are not known by the developers (Szyperski, 2002). Thus, component documentation is a key aspect to development for reuse in general (Silva & Werner, 1996), (Kotula, 1998), (Taulavuori et al., 2004). However the component documentation research area has been neglected (Kotula, 1998). This can be seen in the component-based development methods as well as in the software reuse processes, where this issue is not addressed. However, it is necessary to improve it, "*because without adequate documentation we may become component archeologists, digging through the source code to uncover its functionalities*" (Kotula, 1998).

Most work related to component documentation (Basili & Abd-El-Hafiz, 1996), (Silva & Werner, 1996), (Kotula, 1998), (Taulavuori et al., 2004) are pattern-based approaches. This thesis proposes an activity for component documentation with two important differences: i. it is based on previous works (pattern-based approaches including weak and strong points), a component quality model (Alvaro et al., 2006) and real world experience; ii. it follows some Component Documentation Principles (CDP):

CDP_1. HyperText: if possible, the documentation should be presented as hypertext to enable effective navigation through information;

CDP_2. Embed content in source code: the component documentation should be synchronized with the component source code;

CDP_3 Automation. In order to relieve the workload of the component developers, documentation production and maintenance should be as automated as possible;

CDP_4. Leverage programming languages semantics. The documentation should get advantage of all the information about the component that is inherent in the source code;

CDP_5. Diagrams and Figures. The documentation should include diagrams and figures for clarification purposes. To optimize this principle, the *Automation* (P3) principle should be used whenever possible.

Thus, in order to document software components, the software engineer uses a template[4] composed of five Sections and its related fields. It is important to highlight that as the method for Domain Implementation uses the assets developed in Domain Analysis and Domain Design as input, some parts of the component documentation were already documented in the component specification, and can just be refined. The template is structured as follows (optional fields are written in bold):

Section 1. Basic Information.

- *Name*: the term that characterizes the essence of the component;
- *Problem Description*: describes the problem and the reason why the component has been developed;
- *Purpose*: describes in general what is the purpose of the component;
- *Application Domain*: the application domain in which the component will be used;
- *Keywords*: some basic words to characterize the component;
- *Consequences*: describes the restrictions to be considered and benefits that are derived from the component uses; and
- **Known uses**: describes situations where the component was used in other projects.

Section 2. Detailed Information.

- *Interfaces*: the interface description includes the provided and required interfaces of the component;

[4] This template complements the defined in Appendix F. However, it was not added to avoid misunderstood.

- *Internal Implementation*: includes details about the internal structure of the components derived from the component's class diagram;
- *Language:* describes the language and version used to implement the component; and
- *History version:* includes the initial version and modifications that the component has suffered, and the people who made them.

Section 3. Quality Information.

- *Test Package:* the test package presents links for information such as the test cases defined for the component, the performed tests and their results; and
- **Additional comments***:* this field presents quality attributes related to the component, such as modifiability (how the component can be modified), expandability (how new features can added to the component), and size of the component, in lines of code.

Section 4. Deploy Information.

- *Dependencies*: describes the internal and external dependencies (other frameworks, such as Struts and Tapestry, for example) among the components, in order to allow their use;
- **Source code**: this field contains the files (source code) that compose the software components; and
- *Environment information*: describes information necessary for the deployment of the component, such as operating system and containers for deployment, besides prerequisites such as disk space and memory.

Section 5. Support Information.

- **Install guide** *and Contact*: the component reusers need information on how to install the component and what to do if it does not work as expected. Thus, the component documentation

should have an installation guide, and contact details for the customers. However, the necessity of this guide depends on the size and complexity of the component.

Once implemented and documented, the software components can be tested, certified and made available for reuse. However, these activities are out of scope of this thesis.

7.6. Other Directions in Domain Implementation

The literature does not present many works related to domain implementation. Instead of domain analysis and domain design, however, some directions related to component-based development and software reuse processes can be highlighted.

In (Smaragdakis & Batory, 1998), (Bergner et al., 1999), (Lee et al., 1999), (Griss, 2000), (Crnkovic et al., 2002), (Dogru & Tanik, 2003), (de Jonge, 2003), (Choi, 2004), several approaches for component-based development are presented, involving processes and methodologies based on object-orientation, UML, aspects, and component frameworks. Except for (Smaragdakis & Batory, 1998), (Griss, 2000), these works do not discuss in details how to implement and document software components.

In (Smaragdakis & Batory, 1998), the authors present a method for directly mapping cooperating suites of classes into encapsulated C++ implementations based on nested classes, parameterization, and inheritance. The method is not easy to understand and apply in large-scale. Additionally, detailed steps, inputs, outputs and component documentation issues are not discussed. In (de Jonge, 2003), Jonge proposes a new paradigm: *package-based software development*, which combines fine-grained software reuse and integrated component development. Jonge defines a life-cycle and the steps for this method. However, according to him, package-based software development is not easy to use because both the paradigm and the tools to support it are still subject of research.

Component-Based Development methods, such as Catalysis (D'Souza & Wills, 1998) and UML Components (Cheesman & Daniels, 2000), although

relevant in CBD context, also present gaps when the focus is on component implementation. In Catalysis, for example, the flows among tasks are not concretely defined. Additionally, the method does not deal with component documentation. The same is seen in UML Components.

In the software reuse processes, widely discussed in Chapter 3, in domain implementation, there is a lack of details in component implementation, and documentation is not discussed.

7.7. Chapter Summary

Domain implementation methods and approaches are not given the proper attention by the research community. A possible reason could be that there are already a high number of implementation techniques available today, providing several different options for implementing components, each one with its own benefits in terms of the different quality attributes that a component must have. However, these benefits, especially reusability, could be significantly enhanced when the implementation technology is properly tailored to a domain engineering process, culminating with domain implementation activities that focus on the main concepts and principles behind software reuse.

This Chapter presented the third step of RiDE, the domain implementation, providing a set of principles, activities, and sub-activities that software engineers can keep in mind while performing activities to implement and document software components. The next Chapter will present an experimental study in order to evaluate the viability of the utilization of the process in software reuse projects.

The Experimental Study

"No amount of experimentation can ever prove me right; a single experiment can prove me wrong"

Albert Einstein (1879–1955)
German Physicist

Software has become part of our society and it is found in products ranging from microwaves to space shuttles. It implicates that a vast amount of software has been and is being developed. However, the software development is often a complex task and, as consequence, projects may run over a long period of time and involve many people, due to the complexity of the products that are developed. It implies that the software processes also becomes very complex, since it consists of activities, assets, specific tools, and well-defined roles. Consequently, it is difficult to optimize it or even choose or define an efficient process (Wohlin et al., 2000). On the other hand, organizations are continuously trying to improve their software process in order to improve the products and reduce costs and time-to-market.

Nevertheless, when the improvement proposal has been identified, it is necessary to determine which to introduce, if applicable. Moreover, it is often not possible just to change the existing software process without obtaining more information about the actual effect of the improvement proposal, i.e., it is necessary to evaluate the proposals before making any changes in order to reduce risks. In this context, empirical studies are crucial since the progress in any discipline depends on our ability to understand the basic units necessary to solve a problem (Basili, 1996). Additionally, experimentation provides a

systematic, disciplined, quantifiable, and controlled way to evaluate new theories. It has been used in many fields, e.g., physics, medicine, manufacturing; however, in the software engineering field, this idea started to be explored in the 70s with the work of Victor Robert Basili from University of Maryland (Boehm et al., 2005).

According to Basili (1996), like other disciplines, software engineering requires the same high level approach for evolving the knowledge of the discipline: the cycle of model building, experimentation and teaming, since we cannot rely solely on observation followed by logical thought. It involves an experimental component in order to test or disprove theories, to explore new domains. Thus, we must experiment with techniques to see how and when they really work, to understand their limits and to understand how to improve them.

In this sense, this Chapter presents an experimental study in order to evaluate the viability of RiDE in domain engineering projects. Before discussing the experimental study defined, it is necessary to introduce some definitions[1] to clarify its elements.

8.1. Introduction

When conducting an experiment, the goal is to study the outcome when are varied some of the input variables to a process. According to Wohlin et al. (Wohlin et al., 2000), there are two kinds of variables in an experiment: **independent** and **dependent**.

The variables that are objects of the study, which are necessary to study to see the effect of the changes in the independent variables, are called **dependent variables**. Often there is only one dependent variable in an experiment. All variables in a process that are manipulated and controlled are called **independent variables**.

Ex_1. *It is necessary to study the effect of a new development method on the productivity of the personnel.* The *dependent variable* in the experiment is the productivity. *Independent variables* are, for example, the development method, the experience of the personnel, tool support, and the environment.

[1] The definitions and examples were extracted from (Wohlin et al., 2000) pp. 33-34.

An experiment studies the effect of changing one or more independent variables. Those variables are called **factors**. The other independent variables are controlled at a fixed level during the experiment, or else it would not be possible to determine if the factor or another variable causes the effect. A **treatment** is one particular value of a factor.

Ex_2. The *factor* for the example experiment is the development method, since the goal is to study the effect of changing the method. In this case, two *treatment*s are used for the *factor*: the old and the new development method.

The treatments are being applied to the combination of **objects** and **subjects**. An object can, for example, be a document that will be reviewed with different inspection techniques. The people that apply the treatment are called **subjects**.

Ex_3. The *objects* in the example experiment are the programs to be developed and the *subjects* are the personnel.

At the end, an experiment consists of a set of **tests** where each test is a combination of treatment, subject and object.

Ex_4. A *test* can be that person *N* (*subject*) uses the new development method (*treatment*) for developing program *A* (*object*).

8.2. The Experimental Study

According to Wohlin et al. (Wohlin et al., 2000), the experimental process can be divided into the following main activities: the **definition** is the first step, where the experiment is defined in terms of problem, objective and goals. The **planning** comes next, where the design of the experiment is determined, the instrumentation is considered and the threats to the experiment are evaluated. The **operation** of the experiment follows from the design. In the operational phase, measurements are collected, analyzed and evaluated in the **analysis** and **interpretation**. Finally, the results are presented and packaged in the **presentation** and **package**.

The experimental plan presented follows the model proposed in (Wohlin et al., 2000) which is well-defined and tested in several experimental studies. Additionally, the experiment defined in (Barros, 2001) was used as inspiration.

The definition and planning activities will be described in future tense, showing the logic sequence between the planning and operation.

8.2.1. The Definition

In order to define the experiment, the GQM paradigm (Basili et al., 1994) was used. The GQM is based upon the assumption that for an organization to measure in a purposeful way it must first specify the goals for itself and its projects, then it must trace those goals to the data that are intended to define those goals operationally, and finally provide a framework for interpreting the data with respect to the stated goals.

According to (Basili et al., 1994), the result of the application of the GQM is the specification of a measurement system targeting a particular set of issues and a set of rules for the interpretation of the measurement data. The resulting measurement model is composed of three levels:

- *Goal*: a goal is defined for an object, for a variety of reasons, with respect to various models of quality, from various points of view, relative to a particular environment;
- *Question*: a set of questions is used to characterize the way the assessment/achievement of a specific goal is going to be performed based on some characterizing model; and
- *Metric*: a set of data is associated with every question in order to answer it in a quantitative way.

The next Section presents the experiment definition using the GQM paradigm.

8.2.1.1. Goal

G_1. To analyze *the domain engineering process* for the purpose of *evaluating it* with respect to the *efficiency and difficulties of its use* from the point of view of *researcher* in the context of *domain engineering projects*.

8.2.1.2. Questions

Q_1. Does the domain engineering process generate the domain architecture with low instability?

Q_2. Does the domain engineering process generate the domain architecture with high maintainability?

Q_3. Does the domain engineering process generate components with low complexity?

Q_4. Do the subjects have difficulties to apply the process?

8.2.1.3. Metrics

M_1. *Module Stability*. According to Martin (1994), what it is that makes a design rigid, fragile and difficult to reuse is the interdependency of its modules. A design is rigid if it cannot be easily changed, i.e., a single change begins a cascade of changes of independent modules. Moreover, when the extent of change cannot be predicted by the designer, the impact of the change cannot be estimated. It makes difficult the cost of the changes and this way, the design becomes rigid. Thus, the *Instability* (I) metric will measure the module stability in order to assess it. This metric is defined as (Martin, 1994):

I: *Instability*, where $I = (C_e \div (C_a + C_e))$ and

C_a: *Afferent Coupling*, the number of classes outside this category that depend upon classes within this category

C_e: *Efferent Coupling*, the number of classes inside this category that depend upon classes outside this category

The instability metric has range [0, 1], where $I = 0$ indicates a maximally stable category and $I = 1$ indicates a maximally instable category.

M_2. *Module Maintainability*. Maintainability is desirable both as an instantaneous measure and as a predictor of maintainability over time. Efforts to measure and track maintainability are intended to aid reducing or reversing a system's tendency toward "code entropy" or degraded integrity, and to indicate when it becomes cheaper and/or less risky to redesign/rewrite than to change it

(VanDoren, 1997). In domain engineering projects, this issue is very important since a domain is continuously in evolution with new features or products being developed. Thus, the Maintainability Index (MI) metric will measure a module's maintainability in order to assess it. This metric is defined as (VanDoren, 1997):

`MI = 171 - 5.2 * In(aveV) - 0.23 * aveV(g') - 16.2 * In(aveLOC) + 50 * sin(sqrt(2.4 * perCM))`, where:

`aveV` = average Halstead Volume V per module

`aveV(g')` = average extended cyclomatic complexity per module

`aveLOC` = average count of lines of code (LOC) per module

`perCM` = average percent of lines of comments per module

The metric indicates that modules with a MI less than 65 are difficult to maintain, modules between 65 and 85 have reasonable maintainability and those with MI above 85 have good maintainability.

M_3. *Component Complexity*. A critical issue in software engineering is related to separation of concerns and how to modularize a software system in order to result in modules and components that are well defined, testable and maintainable. During the 70s, McCabe developed a mathematical technique (McCabe, 1976) which provides a quantitative basis for modularization and identifying software modules that will be difficult to test or maintain. In this technique, the complexity measure was presented and used to measure and control the number of paths through a program. This measure is also useful to keep the size of the modules manageable and allow the testing of all the independent paths. Thus, the cyclomatic complexity metric will be used to analyze the complexity of each component developed. The cyclomatic complexity of a module is calculated from a connected graph of the module:

`CC(G) = E - N + p`, where:

`E` = the number of edges of the graph

`N` = the number of nodes of the graph

`P` = the number of connected components

This metric indicates[2] that a unit with cyclomatic complexity between 1 and 10, it is a simple program, without much risk. A value between 11 and 20 represents a more complex program, with moderate risk. Values ranging between 21 and 50 represent a complex program, with high risk. Finally, a complexity greater than 50 is an untestable program and presents a high risk.

We decided to use classic Object Orientation metrics to evaluate the process because they are more well-established after years of experience with case studies and experiments. Reuse-specific metrics, although more suited to this context, still need more experimentation and use (Mascena et al., 2005). Besides, issues such maintainability, stability and complexity have a large influence on the architecture. For example, if a component has low maintainability, it will be probably harder to reuse. So, by measuring these aspects, we are, to some extent, measuring reuse.

M_4. *Difficulty to understand the Process*. In order to identify possible weaknesses and misunderstanding in the process and to define improvements, it is necessary to identify and analyze the difficulties found by users using the process. Thus, three metrics related to difficulty will measure this issue:

D_A: %Subjects that had difficulties in the analysis step and the difficulties distribution

D_D: %Subjects that had difficulties in the design step and the difficulties distribution

D_I: %Subjects that had difficulties in the implementation step and the difficulties distribution

Differently from the other metrics, these metrics were never used before, and thus there are no well-known values for them. Thus, arbitrary values were chosen, based on practical experience and common sense. The practical experience during the development of the process has shown that the most difficult task is design, followed by implementation and analysis. Thus, values above 30% for DD, 20% for DI, and 10% for DA are considered as an indicative that the process, in this particular step, is too difficult and should be improved.

[2] Based on research performed in the Software Engineering Institute (SEI).

8.2.2. The Planning

After the definition of the experiment, the planning is started. The definition determines the foundations for the experiment, the reason for it, while the planning prepares for how the experiment is conducted.

Context. The objective of this study is to evaluate the viability of using the domain engineering process based on a domain engineering project. The domain engineering project will be conducted in a university laboratory with the requirements defined by the experimental staff based on real-world projects. The study will be conducted as *single object study* which is characterized as being a study which examines an object on a single team and a single project (Basili et al., 1985). The subjects of the study will be requested to act as the roles defined in the process (domain analyst, domain expert, project manager, etc). However, during the project, one subject can have more than one role depending of the context, e.g., domain analyst in the beginning and domain designer in next the step. All the subjects will be trained to use the process.

Training. The training of the subjects using the process will be conducted in a classroom at the university. The training will be divided in two steps: in the first one, concepts related to software reuse, variability, component-based development, domain engineering, software product lines, asset repository, software reuse metrics, and software reuse processes will be explained during eleven lectures with two hours each. Next, the domain engineering process will be discussed during three lectures. During the training, the subjects can interrupt to ask issues related to lectures. Moreover, the training will be composed of a set of slides and recommended readings.

Pilot Project. Before performing the study, a pilot project will be conducted with the same structure defined in this planning. The pilot project will be performed by a single subject, who will be trained on how to use the proposed process. For the project, the subjects will use the same material described in this planning, and will be observed by the responsible researcher. In this way, the pilot project will be a study based on observation, aiming to detect problems and improve the planned material before its use.

Subjects. The subjects of the study will act as domain analysts, domain experts, designers, and software engineers as defined in the process. However, during the project, they will also act as project managers, configuration managers and testers in order to assure the project's quality.

Instrumentation. All the subjects will receive a questionnaire (QT1) about his/her education and experience, besides a set of three papers (analysis, design, implementation) containing an example of use and the steps of the process. The material also includes a second questionnaire (QT2) for the evaluation of the subjects' satisfaction using the process. The questionnaires compose the Appendix H.

Criteria. The quality focus of the study demands criteria that evaluate the benefits obtained by the use of the process and the difficulties of the users. The benefits obtained will be evaluated quantitatively through the domain architecture and components, using the stability, maintainability, complexity and difficulty metrics defined earlier. Moreover, the difficulties of the users will also be evaluated using qualitative data from questionnaire QT2.

Null Hypothesis. This is the hypothesis that the experimenter wants to reject with as high significance as possible. In this study, the null hypothesis determines that the use of the process in domain engineering projects does not produce benefits that justify its use and that the subjects have difficulties to apply the process. Thus, according to the selected criteria, the following hypothesis can be defined:

H_o: $\mu_{\text{the process generates the architecture with I >= 0.5}}$

H_o: $\mu_{\text{the process generates the architecture with MI < 85}}$

H_o: $\mu_{\text{the process generates components with CC >= 21}}$

H_o: $\mu_{DA >= 10\%}$

H_o: $\mu_{DD >= 30\%}$

H_o: $\mu_{DI >= 20\%}$

It is important to highlight that even though it is important, the domain analysis step will not be evaluated *formally*, since the literature does not present useful techniques or metrics to measure it.

Alternative Hypothesis. This is the hypothesis in favor of which the null hypothesis is rejected. In this study, the alternative hypothesis determines that the use of the process produces benefits that justify its use. Thus, the following hypothesis can be defined:

H_1: $\mu_{\text{the process generates the architecture with I < 0.5}}$

H_1: $\mu_{\text{the process generates the architecture with MI >= 85}}$

H_1: $\mu_{\text{the process generates components with CC < 21}}$

H_1: $\mu_{DA < 10\%}$

H_1: $\mu_{DD < 30\%}$

H_1: $\mu_{DI < 20\%}$

Independent Variables. The independent variables are the education and the experience of the subjects, collected through the questionnaire QT1, the proposed process and the requirements for the project. This information can be used in the analysis for the formation of blocks.

Dependent Variable. The dependent variables are the quality of the architecture and components. The quality of the architecture will be measured through its stability and maintainability. The quality of the components will be measured by its complexity. All the variables will be measured using the *ratio* scale. According to Wohlin et al. (Wohlin et al., 2000), if there is a meaningful zero value and the ratio between two measures is meaningful, a ratio scale can be used.

Qualitative Analysis. The qualitative analysis aims to evaluate the difficulty of the application of the proposed process and the quality of the material used in the study. This analysis will be performed through questionnaire QT2. This questionnaire is very important because it will allow evaluating the difficulties that the subjects have with the process, evaluating the

provided material and the training material, and improving these documents in order to replicate the experiment in the future. Moreover, this evaluation is important because it can be verified if the material is influencing the results of the study.

Randomization. This technique can be used in the selection of the subjects. Ideally, the subjects must be selected randomly from a set of candidates. The only restriction is that they have the requirements defined in the *Subjects* Section.

Blocking. Sometimes, there is a factor that probably has an effect on the response, but the experimenter is not interested in that effect. If the effect on the factor is known and controllable, is possible to use a design technique called blocking. Blocking is used to systematically eliminate the undesired effect in the comparison among the treatments. In this study, it was not identified the necessity of dividing the subjects into blocks, since the study will evaluate just one factor, which is the use of the process.

Balancing. In some experiments, balancing is desirable because it both simplifies and strengthens the statistical analysis of the data. However, in this study it is not necessary to divide the subjects, since there is only one group.

Internal Validity. The internal validity of the study is defined as the capacity of a new study to repeat the behavior of the current study, with the same subjects and objects with which it was executed (Wohlin et al., 2000). The internal validity of the study is dependent of the number of subjects. This study is supposed to have at least between seven and eight subjects to guarantee a good internal validity.

External Validity. The external validity of the study measures its capability to be affected by the generalization, i.e., the capability to repeat the same study in other research groups (Wohlin et al., 2000). In this study, a possible problem related to the external validity is the subjects' motivation, since some subjects can perform the study without responsibility or without a real interest in performing the project with a good quality as it could happen in an industrial project. The external validity of the study is considered sufficient, since it aims to evaluate the viability of the application of the domain

engineering process. Since the viability is shown, new studies can be planned in order to refine and improve the process.

Construct Validity. The validation of the construction of the study refers to the relation between the theory that is to be proved and the instruments and subjects of the study (Wohlin et al., 2000). In this study, a relatively well known and easily understandable problem domain was chosen to prevent the experienced users in a certain domain to make use of it. Thus, this choice avoids previous experience of making a wrong interpretation of the impact of the proposed process.

Conclusion Validity. This validity is concerned with the relationship between the treatment and the outcome, and determines the capability of the study to generate conclusions (Wohlin et al., 2000). This conclusion will be drawn by the use of descriptive statistic.

8.2.3. The Project used in the Experimental Study

The project used in the experimental study was to perform the domain engineering of the starship game domain using the proposed process. In order to do it, three games in this domain were presented to the subjects as shows Figure 8.1. The subjects had just the executables[3] without any documentation (requirements and design specification, source code, etc). After performing the domain engineering, the subjects were asked to implement one application reusing the developed assets. It is important to highlight that the subjects could analyze other games, consult experts, etc, as part of the process.

Figure 8.1. Applications in the Starship Game Domain.

[3] The games were available on the internet in conjunction with their emulators and documentation.

8.2.4. The Instrumentation

Selection of Subjects. For the execution of the study, M.Sc. students in Software Engineering from the Federal University of Pernambuco, Brazil, were selected. The subjects filled the criteria previously stated in the Section 8.2.2. They were selected by *convenience sampling* (Wohlin et al., 2000) representing a non-random subset from the universe of students from Software Engineering. In convenience sampling, the nearest and most convenient people are selected as subjects.

Data Validation. In this study, descriptive statistics will be used to analyze the data set, since it may be used before carrying out hypothesis testing, in order to better understand the nature of the data and to identify abnormal or false data points (Wohlin et al., 2000).

Instrumentation. Before the experiment can be executed, all experiment instruments must be ready. It includes the experiment objects, guidelines, forms and tools. In this study, the questionnaire QT1 and QT2, in conjunction with the papers about the process were used. The questionnaires presented the subjects' names in order to check additional information or misunderstanding. However, the subjects were notified for the information confidentially. Additionally, the subjects could use any tools and environments for domain analysis, architecture design, and component implementation.

8.2.5. The Operation

Experimental Environment. The experimental study was conducted during part of a M.Sc. and Ph.D. Course in Software Reuse, during April-September 2006, at Federal University of Pernambuco. The experiment was composed of seven subjects and the project was developed in 355 hours, 23 minutes and 57 seconds. In this project were specified 44 features, 33 packages, and 79 classes. Additionally, 5 components and 1 example application were also developed, totalizing 3638 lines of code in Java. Table 8.1 summarizes the data for the components.

Table 8.1. Summary of the Components.

Components	Classes	Lines of Code
Audio	3	208
Graphic	12	552
Movement	9	275
Screen	8	507
Strategy	16	707

Figure 8.2 shows the example application developed reusing the assets developed in the project.

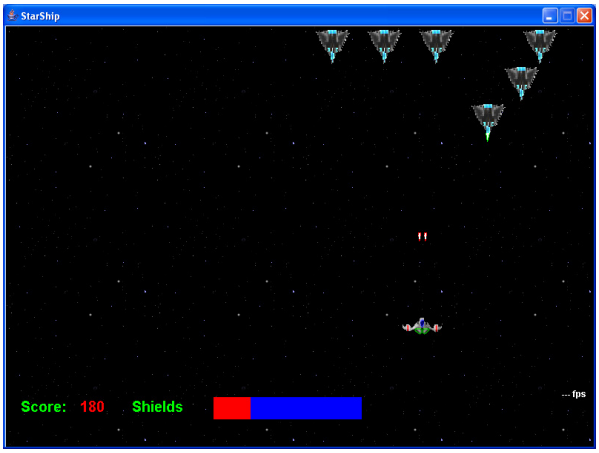

Figure 8.2. Example Application in the Starship Game Domain.

Training. The subjects who used the proposed process were trained before the study began. The training took 28 hours, divided into 14 lectures with two hours each, during the course.

Subjects. The subjects were 7 MS.c. students from the Federal University of Pernambuco. All the subjects had industrial experience in software development (more than one year). Three subjects had participated in industrial projects involving some kind of reuse activity, for instance, component development, framework development, or web services development. All the subjects known at least one domain analysis process (FODA); three subjects had training in conferences on some issues related to software reuse, such as design patterns and component-based development; and finally, two subjects had co-

authored papers involving some aspects of software reuse. Table 8.2 shows a summary of the subjects' profile.

Table 8.2. Subject's Profile in the Experimental Study.

ID	Industrial Projects	Reuse Knowledge	Reuse Training	Reuse Papers
1	-	1 Domain Analysis Process	-	-
2	-	1 Domain Analysis Process	-	-
3	1-2 low complexity 1-2 medium complexity 1-2 high complexity	1 Domain Analysis Process	-	-
4	3-7 low complexity 3-7 medium complexity	1 Domain Analysis Process 1 Domain Design Process	1-2 Conferences	1-2 National and International Papers
5	1-2 medium complexity	1 Domain Analysis Process	-	-
6	-	3 Domain Analysis Processes 1 Domain Design Process	1-2 Conferences	-
7	1-2 low complexity 1-2 medium complexity	1 Domain Analysis 1 Domain Design Process	1-2 Conferences	1-2 National and International Papers

Costs. Since the subjects of the experimental study were students from the Federal University of Pernambuco and the environment for execution was the university's labs and subject's houses (distributed development), the cost for the study was basically planning and operation. The planning for the experimental study took about three months. During this period, it was developed four versions of the planning presented in this thesis.

As previously defined (beginning of the Section - experimental environment), the study was performed in two steps: initially, the subjects were trained in several aspects of software reuse and in the domain engineering

process (April 20, 2006 – June 01, 2006) and after, they performed the domain engineering project between June 08, 2006 - September 12, 2006.

8.2.6. The Analysis and Interpretation

Training Analysis. The training was applied to all the subjects who participated in the study and was composed of a set of slides involving the topics defined in Section 8.2.2. The training was performed in 28 hours as planned. Two subjects considered the training very good (the scale defined was: very good, good, regular, and unsatisfactory); four subjects classified it as good, and one subject as regular. One subject (ID 1) highlighted that the software product lines lecture could show how to build a product line from requirements until the components implementation and product development reusing the assets. For another subject (ID 3), the variability lecture should be revised in order to clarify some basic definitions as variation points, variants, etc. According to two subjects (ID 4, 6), the domain engineering topic could show a complete example from domain analysis until domain implementation in order to improve its understanding. For one subject (ID 4), the repository topic needs to explain how to structure and build an asset repository. At the end, four subjects (ID 2, 4, 6, 7) considered that the metrics lecture should be improved. They suggested two lectures for the topic instead of just one and more real examples including how to collect and analyze the reuse metrics.

Quantitative Analysis. The quantitative analysis was divided in four analyses: instability and maintainability for the architecture, complexity for the components, and the difficulties found in the analysis, design, and implementation steps. The analyses were performed using descriptive statistics.

Instability. After collecting the information about the modules instability, the step of data set reduction started. It is important because errors in the data set can occur either as systematic errors or as *outliers*, which means that the data point is much larger or much smaller than one could expect looking at the other data points (Wohlin et al., 2000). Figure 8.3 shows the instability data graphically.

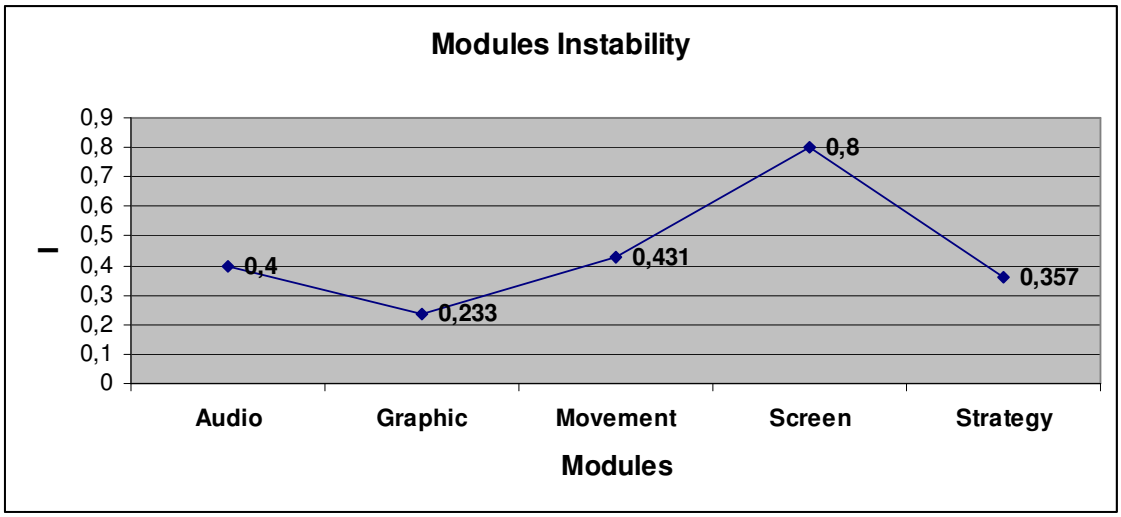

Figure 8.3. Modules Instability Graphic.

As it can be seen in Figure 8.3, *Graphic* and *Screen* presented values low (0.233) and high (0.8), respectively, when compared with the other data points. Thus, these values could be considered as *outliers*. In order to analyze this aspect, a *box plot* graphic can be useful (Fenton & Pfleeger, 1998), since it is recommended to visualize the dispersion and skewedness of samples. Box plots can be made in different ways (Wohlin et al., 2000). In this thesis, the approach defined by Fenton & Pfleeger (1998) was chosen. The main difference among the approaches is how to handle the whiskers. Fenton & Pfleeger proposed to use a value, which is the length of the box, multiplied by *1.5* and added or subtracted from the upper and lower quartiles respectively.

The middle bar in the box is the median. The lower quartile q_1, is the 25% percentile (the median of the values that are less than median), and the upper quartile q_3 is the 75% percentile (the median of the values that are greater than median). The length of the box is $d = q_3 - q_1$.

The tails of the box represent the theoretical bound within all data points are likely to be found if the distribution is normal. The upper tail is $q_3 + 1.5d$ and the lower tail is $q_1 - 1.5d$. Figure 8.4 shows the instability box plot graphic with its information.

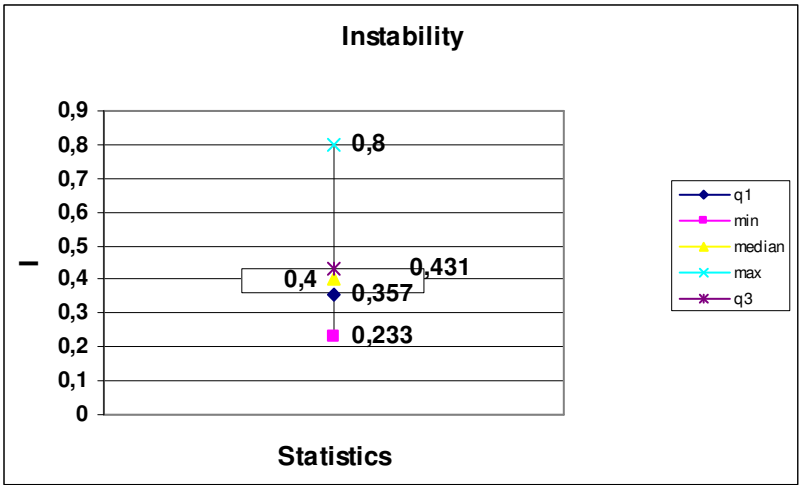

Figure 8.4. Instability Box Plot Graphic.

Values outside the upper and lower tails are the outliers. Figure 8.5 shows the outliers analysis for the instability.

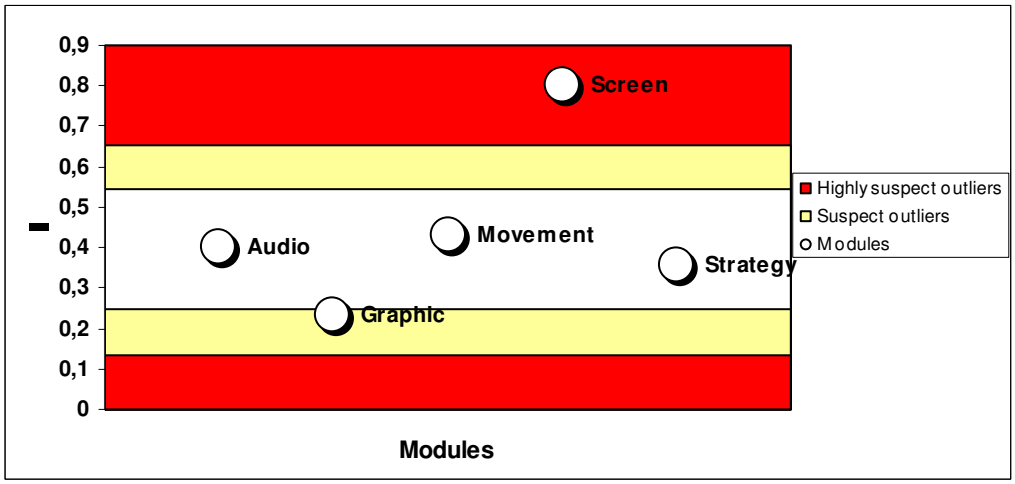

Figure 8.5. Outliers Analysis for the Instability.

As it can be seen in the Figure, *Screen* and *Graphic* represent suspect outliers. When outliers are identified, it is necessary to decide what to do with them: exclude the data or include it in the analysis. In this case, both data were considered, since in the game domain, screen management is the most intensive task that is performed, and thus this module had to be highly coupled with the others, resulting in high instability. On the other hand, *Graphic* is a simple module responsible to represent visual elements for the player such as number of lives, current score and high score, and thus it results in a low instability.

Table 8.3 shows the descriptive statistics with the data collected during the experimental study. The statistics present some relevant information for

analyses. The instability mean (0.444) **rejects** the null hypothesis. It indicates that the process aids in producing the domain architecture with a good stability. Moreover, all the modules, except by *Screen*, present the instability value below the null hypothesis, reinforcing the premises that the process allows to design stable modules.

Table 8.3. Results for the Instability Analysis.

Measure	Instability
Mean	**0.444**
Maximum	**0.8**
Minimum	**0.233**
Standard Deviation	**0.213**
Null Hypothesis	**>= 0.5**

Maintainability. The second analysis consisted in analyzing the maintainability index for the architecture modules. This aspect is very important in a domain, since it is in continuous evolution. Thus, it is important to try to design the domain architecture with a high maintainability index. Figure 8.6 shows the maintainability index data graphically.

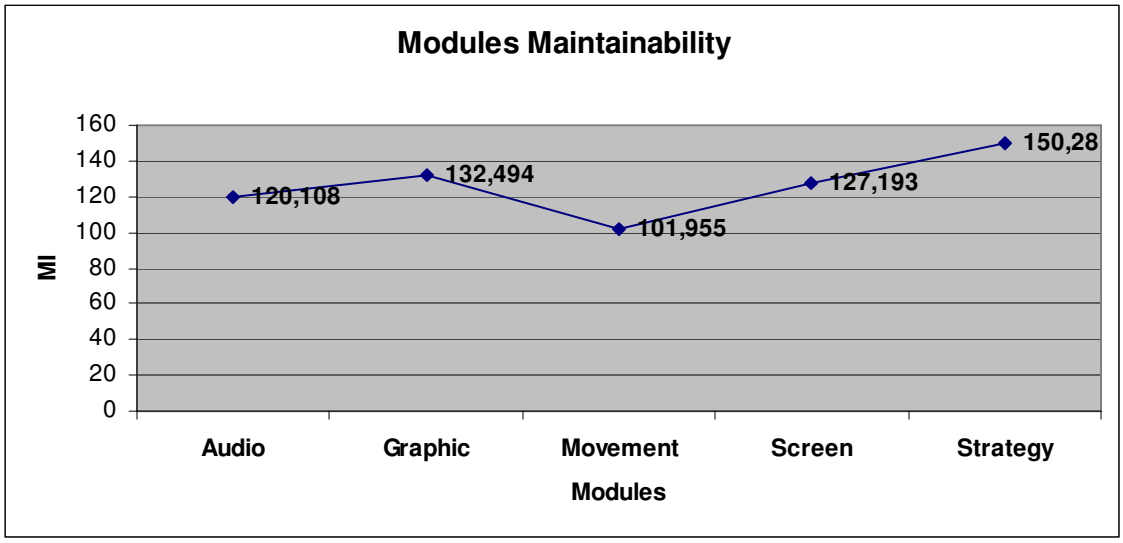

Figure 8.6. Modules Maintainability Graphic.

As it can be seen in the Figure 8.6, in general, the data set does not present a large discrepancy. However, it is necessary a more detailed analysis in order to try to identify possible outliers. Thus, Figure 8.7 and Figure 8.8 show, respectively, the box plot graphic and the outliers analysis for the maintainability.

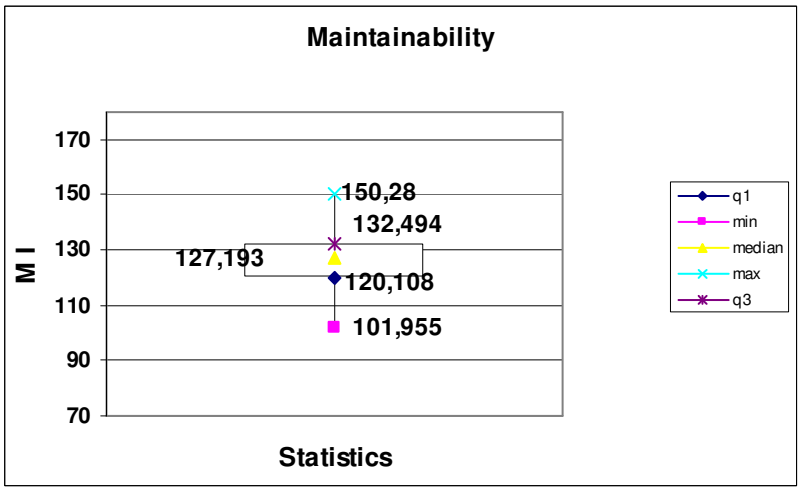

Figure 8.7. Maintainability Box Plot Graphic.

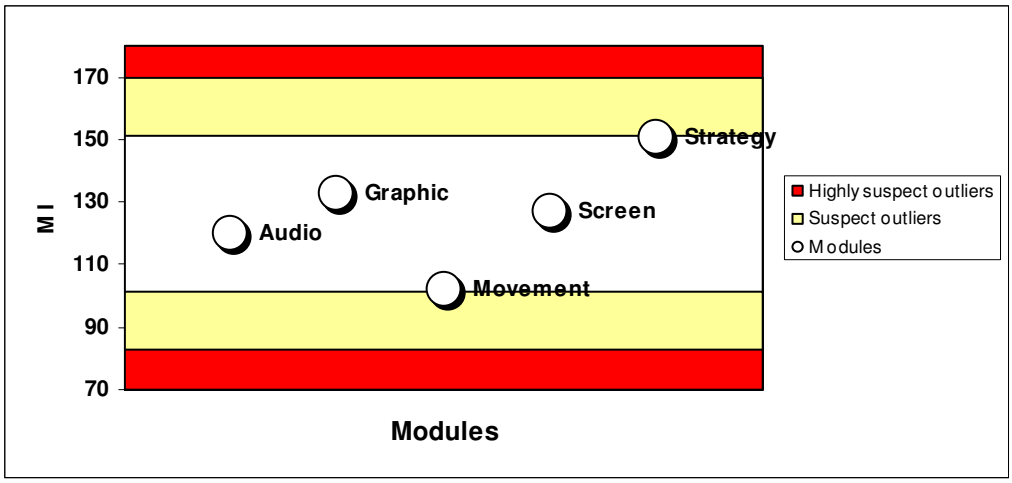

Figure 8.8. Outliers Analysis for the Maintainability.

According to the Figure 8.8, even *Movement* and *Strategy* having values in a different range compared with the others ones, the data set does not present outliers. Since the data set did not present outliers, the next step consisted in analyzing the descriptive statistics.

Table 8.4 shows the descriptive statistics with the data collected during the experimental study. The instability mean (126,406) **rejects** the null hypothesis. It indicates that the process aids in producing the domain architecture with a high maintainability index. Moreover, it is important to highlight that all the modules had MI above 85, what indicates that all of them have a good maintainability and are not difficult to maintain.

Table 8.4. Results for the Maintainability Analysis.

Measure	Maintainability
Mean	**126,406**
Maximum	**150,280**
Minimum	**101,955**
Standard Deviation	**17,643**
Null Hypothesis	**$M_I < 85$**

Complexity. The third analysis consisted in analyzing the components complexity. This aspect is important because it can identify components that will be difficult to test or maintain. Thus, it is important to implement components with low complexity. Figure 8.9 shows the complexity data graphically.

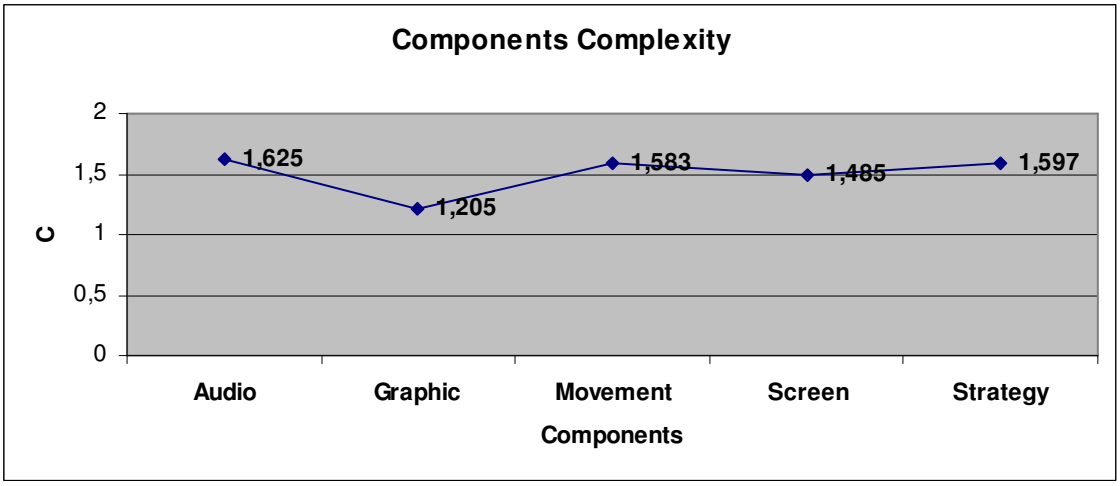

Figure 8.9. Component Complexity Graphic.

According to the Figure, in general, the data present a similar distribution. Figure 8.10 and Figure 8.11 show, respectively, the box plot graphic and the outliers analysis for the complexity.

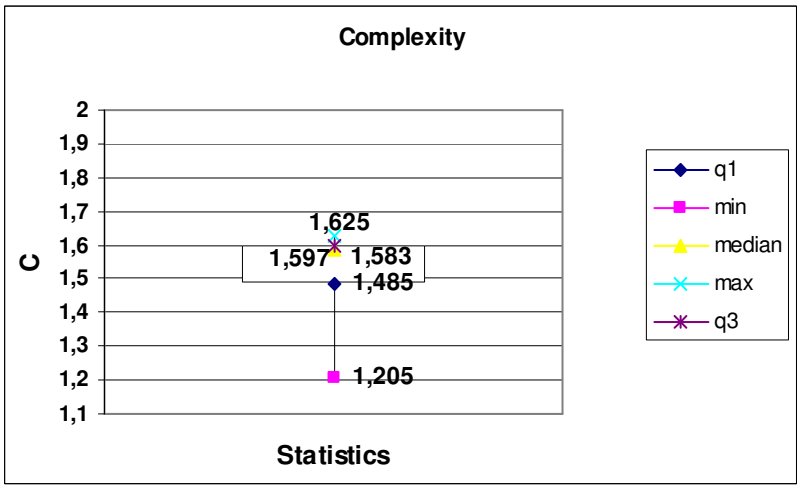

Figure 8.10. Complexity Box Plot Graphic.

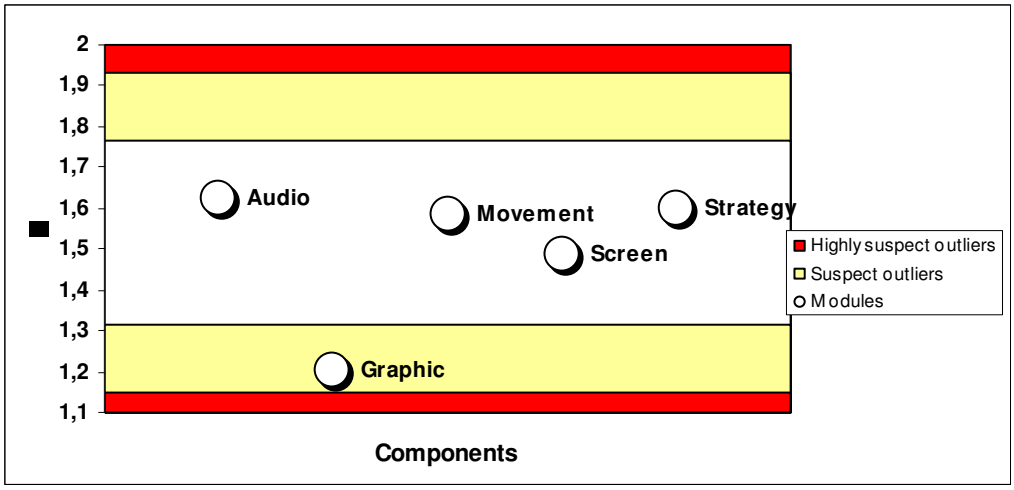

Figure 8.11. Outliers Analysis for the Complexity.

According to Figure 8.11, only *Graphic* presents the complexity in a different range. However, as this component is a simple component, this data was not considered an outlier and it was included in the analysis. The next step consisted in analyzing the descriptive statistics.

Table 8.5 shows the descriptive statistics with the data collected during the experimental study. The complexity mean (1,499) **rejects** the null hypothesis. It indicates that the process aids in producing the components with low complexity. Moreover, it is important to highlight that all the components had CC bellow 21, what indicates that none of them have much risk and are simple.

Table 8.5. Results for the Complexity Analysis.

Measure	Complexity
Mean	**1,499**
Maximum	**1,625**
Minimum	**1,205**
Standard Deviation	**0,173**
Null Hypothesis	**CC >= 21**

Difficulties in the Analysis step. Analyzing the subjects' answers for the difficulties the in analysis step, it was identified that just one subject (ID 2) did not have difficulties in domain analysis step. One subject (ID 7) did not answer this question and this data point was excluded from analysis. On the other hand, five subjects (ID 1, 3, 4, 5, 6) had problems to understand this step, representing approximately 83,3% of the total. This aspect **confirms** the null hypothesis (D_A >= 10%). However, it is necessary to highlight that this value for the null hypothesis was defined without any previous data, since it was the first time that this aspect was analyzed, differently from the other metrics, which had pre-established values. Nevertheless, the next time that the experiment is performed this value can be refined based on this experience, resulting in a more calibrated metric. Figure 8.12 shows the histogram with the distribution density of the found difficulties.

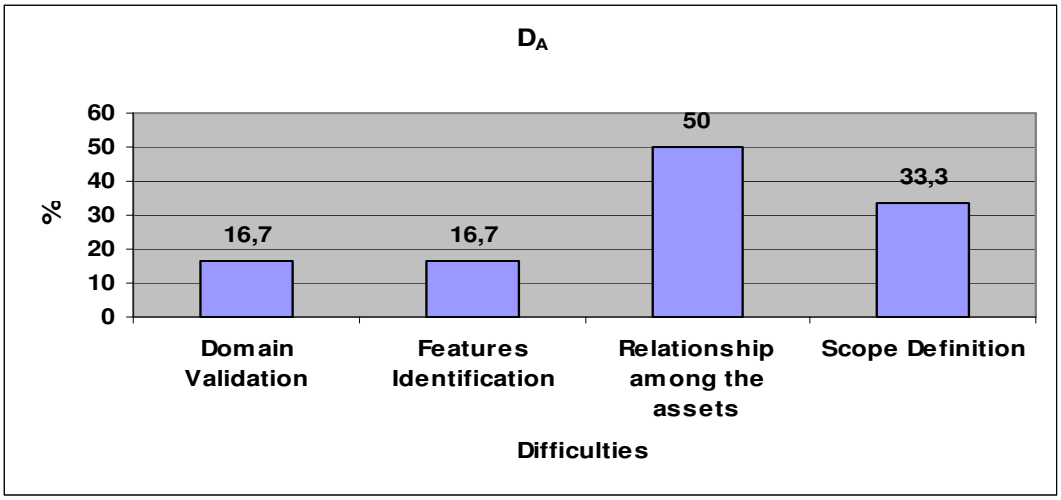

Figure 8.12. Domain Analysis Difficulties Histogram.

According to the Figure, one subject (ID 4) had difficulty to understand the validate domain activity. For another subject (ID 6), the domain analysis step should define guidelines to identify features. Three subjects (ID 3, 5, 6) considered that the domain analysis step should define and show the

relationship among features, requirements, and use cases, since it was a confused aspect during the project execution. At the end, two subjects (ID 1, 6) reported that they had difficulties to understand (ID 6) and use the evaluation functions (ID 1, 6) in order to define the domain scope.

Difficulties in the Design step. After analyzing the subjects' answers for the difficulties in analysis, the same process was performed for the design. It was identified that just one subject (ID 1) did not have difficulties in the domain design step. One subject (ID 7) did not answer this question and this data point was excluded from the analysis. On the other hand, five subjects (ID 2, 3, 4, 5, 6) had problems to understand this step, representing approximately 83,3%. This aspect **confirms** the null hypothesis (D_D >= 30%). However, in the same way as in the analysis step, this value for the null hypothesis was defined without any previous data. Figure 8.13 shows the histogram with the distribution density of the found difficulties.

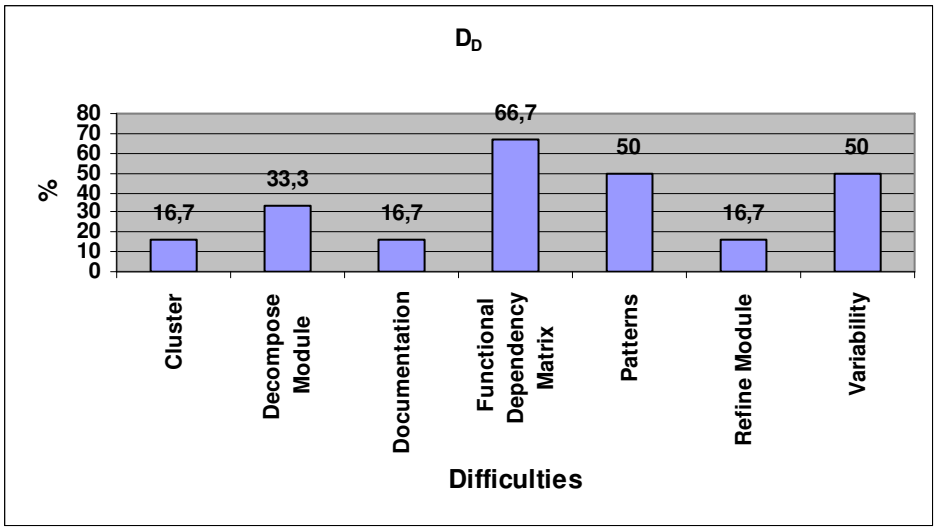

Figure 8.13. Domain Design Difficulties Histogram.

According to the Figure, one subject (ID 6) had difficulty to understand the cluster process and recommended more examples showing how to do it. For two subjects (ID 2, 6), the decompose module activity needs to be better explained because its relationship with the refine module activity was obscure. Moreover, one subject (ID 6) had difficulty to identify the architectural drivers. The same subject reported that the documentation for the design step (paper) has a lack of details and needs to be reviewed in order to improve its

understanding. Four subjects (ID 3, 4, 5, 6) had difficulty to create the functional dependency matrix, and one of them (ID 3) reported that examples can be useful to build it. Another commented aspect was the pattern applicability. For one subject (ID 2), more examples are needed, to show how to represent the variability using the design patterns. Another subject (ID 3) highlighted that it is necessary to define the right moment to use the patterns and recommended examples using the GRASP patterns. At the end, other subject (ID 6) commented that guidelines should be defined in the refine module activity for using the architectural patterns. The same subject had difficulty to identify the architectural drivers for the modules, recommending examples showing how to identify it. Finally, the last issue was related to variability. Three subjects (ID 3, 4, 5) had difficulties to represent the variability in design, especially, to map the variability present in features into classes.

Difficulties in the Implementation step. The last quantitative analysis consisted in analyzing the difficulties in the implementation step. In this process, two subjects (ID 5, 6) reported to have difficulties due to the lack of experience in component implementation. Additionally, two subjects (ID 1, 6) identified the same problem: lack of experience, however, related to OSGi technology. At the end, for one subject (ID 5), the integration between design and implementation needs to be more clarified. Figure 8.14 shows the histogram with the distribution density of the found difficulties.

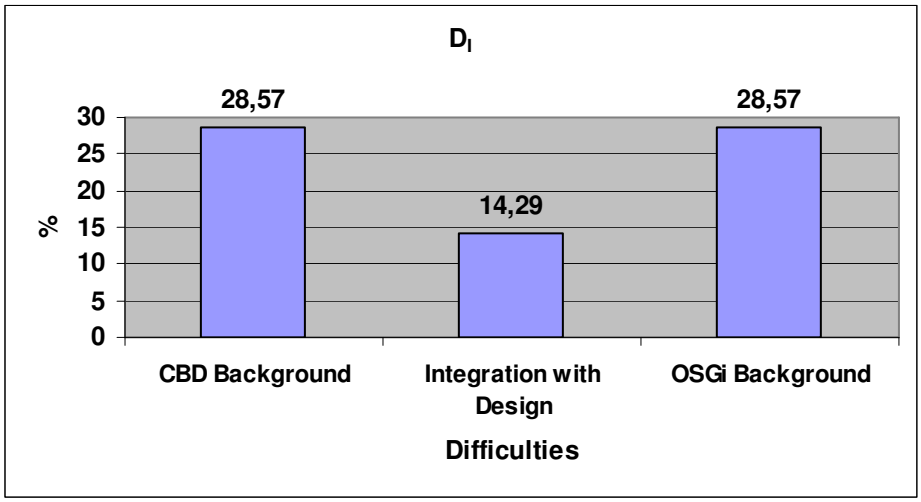

Figure 8.14. Domain Implementation Difficulties Histogram.

The analysis showed that approximately 42.86% of the subjects had difficulties to understand the domain implementation step. This aspect **confirms** the null hypothesis (D_I >= 20%). However, in the same way as in analysis and design, this value for the null hypothesis was defined without any previous data.

Conclusion. Even with the analysis not being conclusive, the experimental study indicates that the process allows designing the domain architecture with a good stability and maintainability. Additionally, the components developed using the process present a low complexity. On the other hand, the aspects related to understanding (difficulties in analysis, design, and implementation) needs to be reviewed and improved. However, with the results identified in the experiment, the values can be calibrated in a more accurate way. Nevertheless, most problems identified by the subjects in terms of difficulties are more related to the provided training than to the process itself. This is further discussed next, in the qualitative analysis.

Qualitative Analysis. After concluding the quantitative analysis for the experiment, the qualitative analysis was performed. This analysis is based on the answers defined for the QT2 presented in Appendix H.

Usefulness of the Process. All the subjects reported that the process was useful to perform the domain engineering project. However, four subjects (ID 1, 2, 5, 6) indicated some improvements in domain analysis; for five subjects (ID 2, 3, 4, 5, 6), some aspects in design should be reviewed; and, finally, six subjects (ID 1, 2, 4, 5, 6, 7) discussed some improvements in domain implementation. All these issues were previously detailed in the sub-sections difficulties in analysis, design, and implementation.

Quality of the Material. Only one subject considered the training insufficient for applying the process. Other three subjects also indicated improvements in lectures related to software product lines, variability and asset repository. Most subjects also complained about the lack of more examples to clarify the different activities of the process, such as the use of evaluation functions and design patterns. Some of these aspects were only related to the background to use the process, but

some issues have influenced on the difficulty of use, as demonstrated in the quantitative analysis.

8.2.7. Lessons Learned

After concluding the experimental study, some aspects should be considered in order to repeat the experiment, since they were seen as limitations of the first execution.

Domain. Two subjects (ID 1, 5) reported that the game domain should be re-discussed since it is a non-trivial domain and often a domain expert is necessary to aid in the process.

Training. Besides the improvements related to the lectures, three subjects (ID 4, 5, 6) highlighted that the training should include a complete and detailed example covering the whole process.

Questionnaires. The questionnaires should be reviewed in order to collect more precise data related to feedback and the process. Moreover, a possible improvement can be to collect it after the iterations during the project avoiding losing useful information by the subjects.

Pilot Project. The pilot project was performed just by one subject. Perhaps, this aspect should be re-analyzed in order to include all the subjects that will execute the experimental study.

Separated Roles. During the project, some subjects were associated to two roles: the defined in the process and others, such as project manager, tester, and configuration manager. This could have a negative impact on the project, mainly related to overworking. Thus, this aspect should be re-analyzed and a possible solution can be an independent team to perform these tasks (management, quality control, testing).

Subjects Skill. The process does not define the skills necessary to each role in the process. Moreover, in this experiment, the roles were defined in an informal way, often allocating the subjects for the roles defined in their jobs. However, this issue should be reviewed in order to be more systematic and reduce risks.

Motivation. As the project was relatively long, it was difficult to keep the subjects' motivation during all the execution. Thus, this aspect should be analyzed in order to try to control it. A possible solution can be to define some checkpoints during the project.

8.3. Chapter Summary

This Chapter presented the definition, planning, operation, analysis and interpretation of the experimental study that evaluated the viability of the domain engineering process. The study analyzed the possibility of subjects using the process to design the domain architecture with good stability and maintainability, and components with low complexity. The study also analyzed the difficulties found in the analysis, design, and implementation steps.

Even with the reduced number of subjects (7), the analysis has shown that the domain engineering process can be viable. It also identified some directions for improvements. However, two aspects should be considered: the study's repetition in different contexts and studies based on observation in order to identify problems and points for improvements.

The next Chapter will present the conclusions of this work, its main contribution and directions for future works.

Conclusions

"I think and think for months and years. Ninety-nine times, the conclusion is false. The hundredth time I am right"

Albert Einstein (1879-1955)
German Physicist

Software productivity has been steadily increasing since 1970, but not enough to close the gap between the demands placed on the software industry and what the state of the practice can deliver. Currently, as software costs continue to represent an increasing share of computer systems costs, and as software faults continue to be responsible for many expensive failures, nothing short of an order-of-magnitude improvement in software quality and development productivity will save the software industry from its perennial state of crisis (Mili et al., 2002).

In this scenario, software reuse has been touted, since the late 1960s, as a practical and realistic way of delivering such improvements related to cost, productivity, and quality. As discussed in Chapter 2, there are several ways of obtaining the benefits related to software reuse, which involves the combination of *technical* and *non-technical* aspects. Technical aspects include the use of specific methods, processes and tools. On the other hand, non-technical aspects involve issues such as organizational structure, metrics, education, management, etc. Among the technical aspects, the software reuse processes have received relevant attention, since the most common way of reusing software involves developing applications within a predefined application domain, where the application typically differs in terms of specific feature

options or internal details, but share a common architecture and draws on a common pool of reusable assets. Thus, Domain Engineering (definition in Chapter 3, Section 3.2.1.) is a key requirement in a reuse process. However, the available reuse processes do not cover the three steps of domain engineering - domain analysis, domain design, and domain implementation - and neither define activities, sub-activities, roles, inputs and outputs of each step in a systematic way.

In this sense, in order to solve the problems identified in the available reuse processes, this thesis presented RiDE: the RiSE process for domain engineering, which defines a systematic way to perform it based on a set of principles, guidelines, inputs, outputs, and roles. The process is based on an extensive review of existing software reuse processes, involving their weak and strong points.

9.1. Research Contributions

The main contributions of this work can be split into the following aspects: **i.** the realization of a detailed survey on the key developments in the software reuse area; **ii.** the presentation of a more specific survey on the state-of-the-art in software reuse processes research; **iii.** the definition and formalization of a domain engineering process which involves: **iv**. domain analysis, **v.** domain design, **vi.** domain implementation; and **vii.** an experimental study which evaluated the viability of the utilization of the process in a domain engineering project.

- **The Key Developments in the Field of Software Reuse**. Initially, the goal was to understand the software reuse research area, its roots, concepts, characteristics, success and failure cases, company reports, myths, inhibitors, among others, in order to obtain a big picture of the area and discover the future road;
- **A Survey on Software Reuse Processes**. Next, based on this detailed study, the software reuse processes area was investigated. Through this study, fifteen software reuse processes were analyzed and offered the base to define an initial set of requirements for an effective software reuse process, such as: *domain engineering*, *application*

engineering, *metrics*, *economics*, *reengineering*, *adaptation*, and *quality*. In this context, the domain engineering issue was more deeply investigated, since it is considered as a key aspect to reuse success, specially when considering software reuse processes;

- **The RiSE process for Domain Engineering**. After concluding this study, the software process area was briefly analyzed in order to define a solid basis for the process, including its foundations, elements, and steps;
- **The Domain Analysis approach**. Next, the first step of the process was proposed: the domain analysis, where, in summary, the main goal is to identify, organize, and document the common and variable features of applications in a domain;
- **The Domain Design approach**. Even with domain analysis being an important step of the domain engineering life cycle, a crucial point was how to design a domain-specific software architecture which could support several applications in the domain, mainly, using as input the assets defined in the previous step. These assets could be the ones produced in the domain analysis approach presented in RiDE, but could also be the output of another domain analysis effort. Thus, the software architecture and component-based development areas were investigated in order to solve this issue in a systematic way, as a method or approach. In this sense, a domain design approach was proposed;
- **The Domain Implementation approach**. At the end, the last step of the process involved the activities to implement and document the reusable assets, in order to be reused in application engineering;
- **An Experimental Study**. In order to evaluate the process viability in domain engineering projects, an experimental study was performed. The study analyzed the process as quantitatively as well as qualitatively and presented findings that, even with the reduced number of subjects for the experiment, the domain engineering process can be viable.

It is important to highlight that the main contribution of this thesis was to define a systematic process to perform domain engineering, which includes

the steps of domain analysis, domain design, and domain implementation, based on a set of activities, sub-activities, inputs, outputs, principles, guidelines, roles, and the sequence of activities among the three steps, where the output of one can be used as input to the next one.

9.2. Related Work

In the literature, some related work could be identified during this research. Chapter 3 presented fifteen software reuse processes which are close to this work. However, the key difference between this work and others is the systematization of the domain engineering process, an extensively discussed subject through this thesis. This helped to reduce the gaps and lack of detail among the steps. A remark is that all the analyses performed were based on published papers, which are sometimes poor on details. However, in order to avoid making misjudgments, subsequent publications of these works were also analyzed to identify improvements, whenever available.

9.3. Future Work

Due to the time constraints imposed on a Ph.D. degree, this work can be seen as an initial climbing towards the full vision for an efficient and effective domain engineering process, and interesting directions remain to improve what was started and to explore new routes. Thus, the following issues should be investigated as future work:

Feature Interaction. During domain analysis, commonalities and variabilities among applications must be analyzed systematically, since commonalities can be exploited to design reusable assets and variabilities can be used for the design of adaptable and configurable assets. Feature-oriented approaches to commonality and variability analysis have been used extensively in domain engineering (Clements & Northrop, 2001). This is mainly due to the fact that the feature-oriented analysis is an effective way of identifying and modeling variabilities among applications in a domain.

In the feature-oriented approach, variable features (optional, alternative, or features) are considered units of variation. A variable feature, if not properly designed, may affect a large part of assets. If features are independent from each

other, each of them can be designed and developed in isolation, and effects of a variation can be localized to the corresponding component. However, as features are usually not independent, a feature variation may cause changes to many components implementing other features (Lee & Kang, 2004).

Feature dependencies have significant implications in software reuse. In the domain engineering context, applications of a domain must be derivable from reusable assets developed during domain engineering. These assets must be designed so that inclusion or exclusion of variable features causes little changes to components implementing other features.

In order to achieve this goal, various dependencies that variable features have with other features must be analyzed thoroughly before designing reusable assets. With this understanding, these assets must be designed so that variations of a feature can be localized to one or a few components.

Although understanding feature dependencies is critical in domain engineering, they have not been analyzed and documented explicitly. In this context, a possible direction is to investigate the feature interaction area in order to define a mechanism to manage this task during the domain analysis design step.

Other Directions in Software Architecture. The domain design step proposed in this thesis is based on two solid concepts: component-based development and design patterns. These areas have shown a strong connection with the software architecture area, however, new directions are starting to arise in the software architecture field such as: Model-Driven Architectures (MDA) (Schmidt, 2006) and Service-Oriented Architecture (SOA) (Gold et al., 2004). We believe that these two directions can define a new route in domain engineering processes focusing on models and services instead of just components as defined in this thesis. Some directions in this sense are being investigated (Muthig & Atkinson, 2002) but not related explicitly to a software reuse process.

Architecture Evaluation. Software architecture is a key asset for any organization that builds complex software-intensive systems. In domain engineering, this issue is still more important, since the architecture should

support not just one, but several applications in a domain. Thus, organizations should analyze their architecture before committing resources to it. According to Kazman et al. (2005), evaluating a software architecture provides an effective means for understanding the tradeoffs involved in a design and in determining the quality attribute characteristics of an architecture design. Even being very important in domain engineering processes this aspect is not considered. Some directions (Etxeberria & Mendieta, 2005) are being investigated but not specifically in a domain engineering process. Thus, it can be an important field to be investigated and integrated in the process. The approach can be to extend a specific method (Dobrica & Niemela, 2002) for the context of domain-specific software architecture or to define a new one.

Architecture Documentation. Software architecture has emerged as an important sub-discipline of software engineering, particularly in the realm of large-system development. According to Clements et al. (2004), architecture can be seen as what makes the sets of parts work together as a successful whole. On the other hand, architecture documentation is what tells developers how to make it so. The software architecture documentation field is an important issue and is growing in importance in the software architecture area. However, this aspect is not being explored in the context of domain-specific software architecture. In this kind of architecture, this aspect is still more critical since the architecture supports the set of applications in the domain and the documentation for the designers is essential. In RiDE, we define some templates which can be useful for documenting the architecture, nevertheless, more systematical research as the described in (Clements et al., 2004) can be useful.

Metrics. Metrics have an important role in software engineering. They allow the engineers to define quality goals, to measure it, and to monitor the accomplishment of these goals. There are a set of software metrics related to different aspects such as design, code, etc (Fenton & Pfleeger, 1998). In this thesis, we used some object-oriented metrics to assess issues such as design, implementation, and maintenance. However, we think that these metrics can be combined with software reuse metrics (Poulin, 1997) in order to define an organizational framework for software reuse projects. This framework can be

used to define the right metrics for using in different phases during the software development life cycle.

Process Modeling Language (PML). The effort placed on the notion of software process has motivated a number of research initiatives (Fuggetta, 2000). One of them is related to the techniques and methods to model software processes and to support their execution (or enactment). Thus, as software processes are complex entities, researchers have created a number of languages and modeling formalisms, the called Process Modeling Languages. According to Fuggetta (2000), PML makes possible to represent in a precise and comprehensive way a number of software process features and facets, i.e., activities, roles, artifacts, and tools. RiDE used the SA language to represent the proposed process. However, another important direction that can be investigated is related to the specification proposed by the OMG: the Software Process Engineering Metamodel (SPEM) (OMG, 2005a). According to the OMG (2005a), the SPEM is a metamodel to define processes and their components. The metamodel has been experimented at least in the Rational Unified Process, DMR Macroscope, IBM's Global Services Method and the Unisys QuadCycle method, however, the limitations, lesson learned and results about it were not discussed. Nevertheless, we believe that it can be useful since it is a standard and the process definition can be imported in different commercial tools due to the interoperability feature though XML Metadata Interchange (XMI) files.

Experimental Study. This thesis presented the definition, planning, operation, analysis, interpretation, presentation and packaging of an experimental study. However, new studies in different contexts, including more subjects and other domains are still necessary in order to calibrate the proposed process and the experimental plan.

Full Process. In Chapter 3, we identified a set of requirements for an efficient software reuse process. We believe that the requirements identified, such as adaptation, application engineering, economic aspects, metrics, quality aspects (test, inspection, certification), and reengineering, in conjunction with the identified future works such as feature interaction, and architectural issues can contribute to define a more complete and systematic process for software reuse.

Domain-Specific Languages (DSL). Domain-specific languages are languages tailored to a specific application domain and their primary contribution is to enable reuse of software assets (Mernik et al., 2005). Among the types of assets that can be reused via a specific DSL are grammars, source code, design specifications, and domain abstractions. In order to enable software reuse, the DSLs can be used as input of an application generator, enabling application engineers to reuse semantic notions embodied in the DSL. In RiDE, the domain abstractions gathered in the domain analysis are used as input to perform the mapping for the domain architecture. Another direction to be investigated can be to achieve the mapping from domain abstraction for a DSL. Next, the DSL can be used to build the design abstractions and finally generates applications. This idea is explored with the denomination of software factories (Greenfield et al., 2004). However, its essence is similar to domain engineering or software product lines, i.e., to build reusable assets.

9.4. Academic Contributions

The knowledge developed during this work resulted in the following publications:

- (Almeida et al., 2004) E. S. Almeida, A. Alvaro, D. Lucrédio, V. C. Garcia, S. R. L. Meira, **RiSE Project: Towards a Robust Framework for Software Reuse**, *IEEE International Conference on Information Reuse and Integration* (IRI), Las Vegas, Nevada, USA, November, 2004, pp. 48-53.
- (Almeida & Meira, 2005) E. S. Almeida, S. R. L. Meira, **Towards a Practical and Efficient Process for Software Reuse**, *15th Workshop for Ph.D. Students in Object-Oriented Systems* (PhDOOS), in conjunction with the *19th European Conference on Object-Oriented Programming* (ECOOP), Lecture Notes in Computer Science (LNCS), Glasgow, Scotland, July, 2005.
- (Almeida et al., 2005a) E. S. Almeida, A. Alvaro, D. Lucrédio, V. C. Garcia, S. R. L. Meira, **A Survey on Software Reuse Processes**, *IEEE International Conference on Information Reuse and Integration* (IRI), Las Vegas, Nevada, USA, August, 2005, pp.66-71.

- (Almeida et al., 2006) E. S. Almeida, J. C. C. P. Mascena, A. P. C. Cavalcanti, A. Alvaro, V. C. Garcia, D. Lucrédio, S. R. L. Meira, **The Domain Analysis Concept Revisited: A Practical Approach**, *9th International Conference on Software Reuse* (ICSR), Lecture Notes in Computer Science (LNCS), Torino, Italy, June, 2006, pp. 43-57.
- (Almeida et al., 2007a) E. S. Almeida, A. Alvaro, V. C. Garcia, L. M. Nascimento, D. Lucrédio, S. R. L. Meira, **Designing Domain-Specific Software Architecture: Towards a New Approach,** *6th Working IEEE/IFIP Conference on Software Architecture* (WICSA), Mumbai, India, January, 2007.
- (Almeida et al., 2007b) E. S. Almeida, E. C. R. Santos, A. Alvaro, V. C. Garcia, D. Lucrédio, R. P. M. Fortes, S. R. L. Meira, **Domain Implementation in Software Product Lines,** *7th Internacional Conference on Composition-Based Software Systems* (ICCBSS), Madrid, Spain, February, 2008.
- (Almeida et al., 2007c) E. S. Almeida, L. B. Lisboa, A. Alvaro, V. C. Garcia, D. Lucrédio, S. R. L. Meira, **An Experimental Study in Domain Engineering,** *IEEE EUROMICRO Conference on Software Engineering and Advanced* Applications (SEAA), Component-Based Software Engineering Track, Lubeck, Germany, August, 2007.
- (Almeida et al., 2007e) E. S. Almeida, A. Alvaro, V. C. Garcia, J. C. C. P. Mascena, V. A. A. Burégio, L. M. Nascimento, D. Lucrédio, S. R. L. Meira, **CRUiSE: Component Reuse in Software Engineering**, C.E.S.A.R e-books, 2007, pp. 202.

9.5. Concluding Remarks

Software reuse is a key aspect for organizations interested in improvements related to productivity, quality and cost reductions. In this context, this work presented the RiSE process for domain engineering. The process was based on an extensive review of the available software reuse processes covering their weak and strong points. The process can be seen as a systematic way to perform domain engineering through a well-defined sequence of steps, inputs, outputs, and guidelines. Additionally, the process was evaluated in a domain engineering project through an experimental study which analyzed it both quantitatively and qualitatively and presented findings that the process can be viable.

Even it being an important contribution for the field, new routes need to be investigated in order to define a more complete process. In this sense, were discussed future directions related to feature interaction, software architecture, metrics, PML, experimental studies, and DSL.

At the end, paraphrasing Martin Luther King, Jr. (Carson, 1998), *we have a dream. A dream where more and more people develop software reusing well-defined assets, with a certified quality, a good documentation in order to reduce costs and time-to-market, while increasing the product final quality*. We believe this thesis is one more step in this long and complex road.

References

"I never desire to converse with a man who has written more than he has read"

Samuel Johnson (1709-1784)
English Lexicographer

(Addy, 1998) E. A. Addy, **A framework for performing verification and validation in reuse-based software engineering**, *Annals of Software Engineering*, Vol. 05, January, 1998, pp. 279-292.

(Alexander et al., 1977) C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. F. King, S. Angel, **A Pattern Language: Towns, Buildings, Construction**, Oxford University Press, 1977, pp. 1216.

(Almeida et al., 2004) E. S. Almeida, A. Alvaro, D. Lucrédio, V. C. Garcia, S. R. L. Meira, **RiSE Project: Towards a Robust Framework for Software Reuse**, *IEEE International Conference on Information Reuse and Integration* (IRI), Las Vegas, Nevada, USA, November, 2004, pp. 48-53.

(Almeida & Meira, 2005) E. S. Almeida, S. R. L. Meira, **Towards a Practical and Efficient Process for Software Reuse**, *15th Workshop for Ph.D. Students in Object-Oriented Systems* (PhDOOS), in conjunction with the *19th European Conference on Object-Oriented Programming* (ECOOP), Lecture Notes in Computer Science (LNCS), Glasgow, Scotland, July, 2005.

(Almeida et al., 2005a) E. S. Almeida, A. Alvaro, D. Lucrédio, V. C. Garcia, S. R. L. Meira, **A Survey on Software Reuse Processes**, *IEEE*

Internacional Conference on Information Reuse and Integration (IRI), Las Vegas, Nevada, USA, August, 2005, pp.66-71.

(Almeida et al., 2006) E. S. Almeida, J. C. C. P. Mascena, A. P. C. Cavalcanti, A. Alvaro, V. C. Garcia, D. Lucrédio, S. R. L. Meira, **The Domain Analysis Concept Revisited: A Practical Approach**, *9th International Conference on Software Reuse* (ICSR), Lecture Notes in Computer Science (LNCS), Torino, Italy, June, 2006, pp. 43-57.

(Almeida et al., 2007) E. S. Almeida, A. Alvaro, D. Lucrédio, V. C. Garcia, S. R. L. Meira, **Key Development in the Field of Software Reuse**, *ACM Computing Surveys*, 2007, ***in evaluation***.

(Almeida et al., 2007a) E. S. Almeida, A. Alvaro, V. C. Garcia, L. M. Nascimento, D. Lucrédio, S. R. L. Meira, **Designing Domain-Specific Software Architecture: Towards a New Approach,** *6th Working IEEE/IFIP Conference on Software Architecture* (WICSA), Mumbai, India, January, 2007.

(Almeida et al., 2007b) E. S. Almeida, E. C. R. Santos, A. Alvaro, V. C. Garcia, D. Lucrédio, R. P. M. Fortes, S. R. L. Meira, **Domain Implementation in Software Product Lines,** *7th Internacional Conference on Composition-Based Software Systems* (ICCBSS), Madrid, Spain, February, 2008.

(Almeida et al., 2007c) E. S. Almeida, L. B. Lisboa, A. Alvaro, V. C. Garcia, D. Lucrédio, S. R. L. Meira, **An Experimental Study in Domain Engineering,** *IEEE EUROMICRO Conference on Software Engineering and Advanced* Applications (SEAA), Component-Based Software Engineering Track, Lubeck, Germany, August, 2007.

(Almeida et al., 2007d) E. S. Almeida, A. Alvaro, V. C. Garcia, D. Lucrédio, S. R. L. Meira, **A Systematic Approach for Domain Engineering,** *11th International Software Product Line Conference* (SPLC), Kyoto, Japan, September, 2007, ***in evaluation***.

(Almeida et al., 2007e) E. S. Almeida, A. Alvaro, V. C. Garcia, J. C. C. P. Mascena, V. A. A. Burégio, L. M. Nascimento, D. Lucrédio, S. R. L.

Meira, **CRUiSE: Component Reuse in Software Engineering**, C.E.S.A.R e-books, 2007, pp. 202.

(Alvaro et al., 2006) A. Alvaro, E. S. Almeida, S. R. L. Meira, **A Software Component Quality Model: A Preliminary Evaluation**, *32nd IEEE EUROMICRO Conference on Software Engineering and Advanced* Applications (SEAA), Component-Based Software Engineering Track, Cavtat/Dubrovnik, Croatia, August, 2006, pp. 28-35.

(Ambriola, 1997) V. Ambriola, R. Conrado, A. Fuggetta, **Assessing Process-Centered Software Engineering Environments**, *IEEE Transactions on Software Engineering and Methodology* (TOSEM), Vol. 06, No. 03, July, 1997, pp. 283-328.

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

(Anastasopoulos & Gacek, 2001) M. Anastasopoulos, C. Gacek, **Implementing Product Line Variabilities**, *Symposium on Software Reusability* (SSR), Toronto, Canada, May, 2001, pp. 109-117.

(Aoyama et al., 2001) M. Aoyama, **CBSE in Japan and Asia**, in Component-Based Software Engineering: Putting the Pieces Together, G. T. Heineman, B. Councill, Addison-Wesley, 2001, pp. 818.

(Arango, 1989) G. Arango, **Domain analysis: from art form to engineering discipline**, *5th International Workshop on Software specification and design*, Pittsburgh, Pennsylvania, USA, May, 1989, pp. 152-159.

(Atkinson et al., 2000) C. Atkinson, J. Bayer, D. Muthig, **Component-Based Product Line Development: The KobrA Approach**, *First Software Product Line Conference (SPLC)*, Kluwer International Series

in Software Engineering and Computer Science, Denver, Colorado, USA, August, 2000, pp.19.

(Barros, 2001) M. O. Barros, **Project Management based on Scenarios: A Dynamic Modeling and Simulation Approach** (*in portuguese*), Ph.D. Thesis, Federal University of Rio de Janeiro, December, 2001, pp. 249.

(Basili et al., 1985) V. R. Basili, R. W. Selby, D. H. Hutchins, **Experimentation in Software Engineering**, *IEEE Transactions on Software Engineering*, Vol. 12, No. 07, July, 1986, pp. 733-743.

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

(Basili et al., 1994) V. R. Basili, G. Caldiera, H. D. Rombach, **The Goal Question Metric Approach**, *Encyclopedia of Software Engineering*, Vol. II, September, 1994, pp. 528-532.

(Basili et al., 1995) V. R. Basili, M. Zelkowitz, F. McGARRY, J. Page, S. Waligora, R. Pajerski, **SEL's Software Process Improvement Program,** *IEEE Software*, Vol. 12, No. 06, November, 1995, pp. 83-87.

(Basili, 1996) V. R. Basili, **The Role of Experimentation in Software Engineering: Past, Present, and Future**, *18th International Conference on Software Engineering* (ICSE), Berlin, Germany, March, 1996, pp. 442-449.

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

(Basili & Abd-El-Hafiz, 1996) V. R. Basili, S. K. Abd-El-Hafiz, **A Method for Documenting Code Components**, *Journal of Systems and Software* (JSS), Vol. 34, No. 02, August, 1996, pp. 89-104.

(Bass et al., 2000) L. Bass, C. Buhman, S. Dorda, F. Long, J. Robert, R. Seacord, K. Wallnau, **Market Assessment of Component-Based Software Engineering**, Software Engineering Institute (SEI)*, Technical Report*, Vol. 01, May, 2000, pp. 41.

(Bass et al., 2003) L. Bass, P. Clements, R. Kazman, **Software Architecture in Practice**, 2nd Edition, Addison-Wesley, 2003, pp. 560.

(Baster et al., 2001) G. Baster, P. Konana, J. E. Scott, **Business Components: A case study of bankers trust Australia limited**, *Communications of the ACM*, Vol. 44, No. 05, May, 2001, pp. 92-98.

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

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

(Bayer et al., 1999a) J. Bayer, D. Muthig, T. Widen, **Customizable Domain Analysis**, *First International Symposium on Generative and Component-Based Software Engineering* (GCSE), Erfurt, Germany, September, 1999, pp. 178-194.

(Bergey et al., 1999) J. Bergey, D. Smith, S. Tilley, N. Weiderman, S. Woods, **Why Reengineering Projects Fail**, Software Engineering Institute (SEI), *Technical Report*, April, 1999, pp. 37.

(Bergner et al., 1999) K. Bergner, A. Rausch, M. Sihling, A. Vilbig, **Componentware Methodology and Process**, *International Workshop on Component-Based Software Engineering* (CBSE), Los Angeles, USA, May, 1999, pp. 05.

(Blois et al., 2005) A. P. Blois, C. M. L. Werner, K. Becker, **Towards a Components Grouping Technique within a Domain Engineering Process**, *31st IEEE EUROMICRO Conference on Software Engineering and Advanced Applications* (EUROMICRO-SEAA), Component-Based Software Engineering Track, Porto, Portugal, August-September, 2005, pp. 18-27.

(Boehm, 1988) B. W. Boehm, **A Spiral Model of Software Development and Enhancement**, *IEEE Computer*, Vol. 21, No. 05, May, 1988, pp. 61-72.

(Boehm et al., 2005) B. W. Boehm, H. D. Rombach, M. V. Zelkowitz, **Foundations of Empirical Software Engineering: The Legacy of Victor R. Basili**, Springer, 2005, pp. 431.

(Boehm, 2006) B. W. Boehm, **A View of 20th and 21st Century Software Engineering,** *28th International Conference on Software Engineering* (ICSE), Shanghai, China, May, 2006, pp. 12-29.

(Bosch, 2000) J. Bosch, **Design and Use of Software Architecture,** Addison-Wesley, 2000, pp. 354.

(Briand, 2003) L. C. Briand, **Software Documentation: How Much is Enough?**, *7th European Conference on Software Maintenance and Reengineering* (CSMR), Benevento, Italy, March, 2003, pp. 13-13.

(Bronsard et al., 1997) F. Bronsard, D. Bryan, W. Kozaczynski, E. S. Liongosari, J. Q. Ning, Á. Ólafsson, J. W. Wetterstrand, **Toward Software Plug-an-Play**, *Symposium on Symposium on Software Reusability* (SSR), Boston, USA, May, 1997, pp.19-29.

(Broy, 2006) M. Broy, **The 'Grand Challenge' in Informatics: Engineering Software-Intensive Systems**, *IEEE Computer*, Vol. 39, No. 10, October, 2006, pp. 72-80.

(Brown & Wallnau, 1998) A. W. Brown, K. C. Wallnau, **The Current State of CBSE,** *IEEE Software*, Vol. 15, No. 05, September/October, 1998, pp. 37-46.

(Burégio, 2006) V. A. A. Burégio, **Specification, Design, and Implementation of a Reuse Repository**, M.Sc. Dissertation, Federal University of Pernambuco, August, 2006.

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

(Carson, 1998) C. Carson, **The Autobiography of Martin Luther King, Jr.,** Warner Books, 1998, pp. 400.

(Cheesman & Daniels, 2000) J. Cheesman, J. Daniels, **UML Component A Simple Process for Specifying Component-Based Software**, Addison-Wesley, 2000, pp. 208.

(Chen, 2002) F. Chen, Q. Wang, H. Mei, F. Yang, **An Architecture-Based Approach for Component-Oriented Development**, *26th International Computer Software and Applications Conference* (COMPSAC), Oxford, England, August, 2002, pp. 450-458.

(Choi et al., 2004) S.W. Choi, S.H. Chang, S.D. King, **A Systematic Methodology for Developing Component Frameworks**, *7th International Conference in Fundamental Approaches to Software Engineering*, Barcelona, Spain, March, 2004, pp. 359-373.

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

(Clements et al., 2004) P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, J. Stafford, **Documenting Software Architectures: Views and Beyond**, Addison-Wesley, 2004, pp. 512.

(Coplin et al., 1998) J. Coplin, D. Hoffman, D. Weiss, **Commonality and Variability in Software Engineering**, *IEEE Software*, Vol. 15, No. 06, November-December, 1998, pp. 37-45.

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

(Crnkovic & Larsson, 2000) I. Crnkovic, M. Larsson, **A Case Study: Demands on Component-Based Development**, *22nd International Conference on Software Engineering* (ICSE), Limerick, Ireland, June, 2000, pp. 23-31.

(Crnkovic et al., 2002) I. Crnkovic, B. Hnich, T. Jonsson, Z. Kiziltan, **Specification, Implementation, and Deployment of Components**, *Communications of the ACM*, Vol. 45, No. 10, October, 2002, pp. 35-40.

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

(Dahl, et al., 1972) O. J. Dahl, E. W. Dijkstra, C. A. R. Hoare, **Structured Programming**, Academic Press, 1972, pp. 220.

(Debaud & Schmid, 1999) J. M. Debaud, K. Schmid, **A Systematic Approach to Derive the Scope of Software Product Lines**, *International Conference on Software Engineering* (ICSE), Los Angeles, US, May, 1999, pp. 34-43.

(de Jonge, 2003) M. de Jonge, **Package-Based Software Development**, *29th IEEE EUROMICRO Conference on Software Engineering and Advanced Applications* (EUROMICRO-SEAA), Component-Based Software Engineering Track, Belek-Antalya, Turkey, September, 2003, pp. 76-85.

(Dikel et al., 1997) D. Dikel, D. Kane, S. Ornburn, W. Loftus, J. Wilson, **Applying Software Product-Line Architecture**, *IEEE Computer*, Vol. 30, No. 08, August, 1997, pp. 49-55.

(Dobrica & Niemela, 2002) L. Dobrica, E. Niemela, **A Survey on Software Architecture Analysis Methods**, *IEEE Transactions on Software Engineering*, Vol. 28, No. 07, July, 2002, pp. 638-653.

(Dogru & Tanik, 2003) A. H. Dogru, M. M. Tanik, **A Process Model for Component-Oriented Software Engineering**, *IEEE Software*, Vol. 20, No. 02, January, 2003, pp. 34-41.

(D'Souza & Wills, 1998) D. F. D'Souza, A. C. Wills, **Objects, Components and Frameworks with UML: The Catalysis Approach**, Addison-Wesley, 1998, pp. 816.

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

(Etxeberria & Mendieta, 2005) L. Etxeberria, G. S. Mendieta, **Product-Line Architecture: New Issues for Evaluation**, *9th International*

Conference on Software Product Lines (SPLC), Rennes, France, September, 2005, pp. 174-185.

(Ezran et al., 2002) M. Ezran, M. Morisio, C. Tully, **Practical Software Reuse**, Springer, 2002, pp. 374.

(Fayad, 1997) M. E. Fayad, **Software Development Process: A Necessary Evil**, *Communications of the ACM*, Vol. 40, No. 09, September, 1997, pp. 101-103.

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

(Fenton & Pfleeger, 1998) N. Fenton, S. L. Pfleeger, **Software Metrics: A Rigorous & Practical Approach**, Course Technology, 2nd Edition, 1998, pp. 656**.**

(Fichman & Kemerer, 1997) R. G. Fichman, C. F. Kemerer, **Object Technology and Reuse: Lessons from Early Adopters**, *IEEE Computer*, Vol. 30, No. 10, October, 1997, pp. 47-59.

(Frakes, 1993) W. B. Frakes, **Software Reuse as Industrial Experiment**, *American Programmer*, Vol. 06, No. 08, November, 1993, pp. 27-33.

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

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

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

(Frakes et al., 1998) W. B. Frakes, R. Prieto-Diaz, C. Fox, **DARE: Domain Analysis and reuse environment**, *Annals of Software Engineering*, Vol. 05, January, 1998, pp. 125-141.

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

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

(Freeman, 1987) P. Freeman, **Reusable software engineering concepts and research directions**, *Tutorial Software Reusability*, IEEE Computer Society Press, 1987.

(Fuggetta, 2000) A. Fuggetta, **Software Process: A Roadmap**, *22nd International Conference on Software Engineering* (ICSE), *The Future of Software Engineering* (FOSE), Limerick, Ireland, 2000, June, 2000, pp. 25-34.

(Gamma et al., 1995) E. Gamma, R. Helm, R. Johnson, J. Vlissides, **Design Patterns: Elements of Reusable Object-Oriented Software**, Addison-Wesley, 1995, pp. 395.

(Garcia et al., 2006) V. C. Garcia, D. Lucrédio, F. A. Durão, E. C. R. Santos, E. S. Almeida, R. P. M. Fortes, S. R. L. Meira, **From Specification to the Experimentation: A Software Component Search Engine Architecture**, 9th *International Symposium on Component-Based Software Engineering* (CBSE), Lecture Notes in Computer Science (LNCS), Springer-Verlag, Sweden, July, 2006, pp. 82-97.

(Garcia et al., 2006a) V. C. Garcia, D. Lucrédio, E. S. Almeida, R. P. M. Fortes, S. R. L. Meira, **Towards a Code Search Engine based on the State-of-the-Art and Practice**, *13th IEEE Asia-Pacific Software Engineering Conference* (APSEC), Component-Based Development Track, Bangalore, India, December, 2006, pp. 61-68.

(Garlan, 1995) D. Garlan, **Research Directions in Software Architecture**, *ACM Computing Surveys*, Vol. 27, No. 02, June, 1995, pp. 257-261.

(Ghosh, 2006) P. Ghosh, **Java Component Development: A Conceptual Framework**, ONJava.com, Available in

http://www.onjava.com/pub/a/onjava/2005/03/23/components.html , Consulted in December 2006.

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

(Glass, 2002) R. L. Glass, **Facts and Fallacies of Software Engineering**, Addison-Wesley, 2002, pp. 224.

(Gold et al., 2004) N. Gold, A. Mohan, C. Knight, M. Munro, **Understanding Service-Oriented Software**, *IEEE Software*, Vol. 21, No. 02, March/April, 2004, pp. 71-77.

(Gomaa, 2005) H. Gomaa, **Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures**, Addison-Wesley, 2005, pp. 701.

(Greenfield et al., 2004) J. Greenfield, K. Short, S. Cook, S. Kent, **Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools**, Wiley, 2004, pp. 666.

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

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

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

(Griss, 2000) M. L. Griss, **Implementing Product-Line Features with Component Reuse**, *6th International Conference on Software Reuse* (ICSR), Vienna, Austria, June, 2000, pp. 137-152.

(Heineman & Councill, 2001) G. T. Heineman, W. T. Councill, **Component-Based Software Engineering**, Addison-Wesley, 2001, pp. 818.

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

(Henry & Kafura, 1981) S. Henry, D. Kafura, **Software Structure Metrics Base don Information Flow**, *IEEE Transactions on Software Engineering*, Vol. 07, No. 05, September, 1981, pp. 510-518.

(Hofmeister et al., 1999) C. Hofmeister, R. Nord, D. Soni, **Applied Software Architecture,** Addison-Wesley, 1999, pp. 397.

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

(Jacobson & Lindstrom, 1991) I. Jacobson, F. Lindstrom, **Reengineering of old systems to an object-oriented architecture**, *Sixth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications* (OOPSLA), Phoenix, Arizona, USA, October, 1991. pp. 340–350.

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

(Jacobson et al., 1997a) I. Jacobson, M. L. Griss, P. Jonsson, **Making the Reuse Business Work**, *IEEE Computer*, Vol. 30, No. 10, October, 1997, pp. 36-42.

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

(Jain & Chalimeda, 2001) H. Jain, N. Chalimeda, **Business Component Identification – A Formal Approach**, *5th International Enterprise Distributed Object Computing Conference* (EDOC), Seattle, USA, September, 2001, pp. 183-187.

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

(John et al., 2006) I. John, J. Knodel, T. Lehner, D. Muthig, **A Practical Guide to Product Line Scoping**, *10th International Software*

Product Line Conference (SPLC), Baltimore, Maryland, August, 2006, pp. 03-12.

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

(Kang et al., 1990) K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, A. S. Peterson, **Feature-Oriented Domain Analysis (FODA) Feasibility Study**, Software Engineering Institute (SEI), *Technical Report*, November, 1990, pp. 161.

(Kang et al., 1998) K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, M. Huh, **FORM: A Feature-Oriented Reuse Method with domain-specific reference architectures**, *Annals of Software Engineering Notes*, Vol. 05, No. 00, Janeiro, 1998, pp. 143-168.

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

(Kazman et al., 1994) R. Kazman, L. J. Bass, M. Webb, G. D. Abowd, **SAAM: A Method for Analyzing the Properties of Software Architecture**, *16th International Conference on Software Engineering* (ICSE), Sorrento, Italy, May, 1994, pp. 81-90.

(Kazman et al., 2005) R. Kazman, L. Bass, M. Klein, T. Lattanze, L. M. Northrop, **A Basis for Analyzing Software Arhchitecture Analysis Methods**, *Software Quality Journal*, Vol. 13, No. 04, December, 2005, pp. 329-355.

(Keepence & Mannion, 1999) B. Keepence, M. Mannion, **Using Patterns to Model Variability in Product Families**, *IEEE Software*, Vol. 16, No. 04, July/August, 1999, pp. 102-108.

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

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

(Kim et al., 2003) M. Kim, H. Yang, S. Park, **A Domain Analysis Method for Software Product Lines Based on Scenarios, Goals and Features**, *10th Asia-Pacific Software Engineering Conference* (APSEC), Chiang Mai, Thailand, December, 2003, pp. 126-136.

(Kim & Chang, 2004) S. D. Kim, S. H. Chang, **A Systematic Method to Identify Software Components**, *11th Asia-Pacific Software Engineering Conference* (APSEC), Seoul, South Korea, December, 2004, pp. 92-98.

(Kotula, 1998) J. Kotula, **Using Patterns To Create Component Documentation**, *IEEE Software*, Vol. 15, No. 02, March/April, 1998, pp. 84-92.

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

(Krueger, 2002) C. W. Krueger, **Practical Strategies and Techniques for Adopting Software Product Lines**, *7th International Conference on Software Reuse* (ICSR), Austin, Texas, April, 2002, pp. 349-350.

(Kruchten, 1995) P. Kruchten, **The 4 + 1 View Model of Architecture**, *IEEE Software*, Vol. 12, No. 06, November, 1995, pp. 45-50.

(Kruchten et al., 2006) P. Kruchten, H. Obbink, J. Stafford, **The Past, Present, and Future of Software Architecture**, *IEEE Software*, Vol. 23, No. 02, March/April, 2006, pp. 22-30.

(Lee et al., 1999) S.D. Lee, Y.J. Yang, E.S. Cho, S.D. Kim, S.Y. Rhew, **COMO: A UML-Based Component Development Methodology**, *6th Asia-Pacific Software Engineering Conference* (APSEC), Takamatsu, Japan, December, 1999, pp. 54-61.

(Lee et al., 2002) K. Lee, K. C. Kang, J. Lee, **Concepts and Guidelines of Feature Modeling for Product Line Software Engineering**, *7th*

International Conference on Software Reuse (ICSR), Austin, Texas, US, April, 2002, pp. 62-77.

(Lee & Kang, 2004) K. Lee, K. C. Kang, **Feature Dependency Analysis for Product Line Component Design**, *8th International Conference on Software Reuse* (ICSR), Madrid, Spain, July, 2004, pp. 69-85.

(Leite, 1994) J. C. S. P. Leite, **Draco-PUC: A Technology Assembly for Domain Oriented Software Development**, *3rd International Conference on Software Reuse* (ICSR), Rio de Janeiro, Brazil, November, 1994, pp. 94-100.

(Lethbridge et al., 2003) T. C. Lethbridge, J. Singer, A. Forward, **How Software Engineers Use Documentation: The State of the Practice**, *IEEE Software*, Vol. 20, No. 06, November/December, 2003, pp. 35-39.

(Leveson & Turner, 1993) N. G. Leveson, C. S. Turner, **An Investigation of the Therac-25 Accidents**, *IEEE Computer*, Vol. 26, No. 07, July, 1993, pp. 18-41.

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

(Lim, 1998) W. C. Lim, **Managing Software Re-use**, Prentice Hall PTR, 1998, pp. 480.

(Lucrédio et al., 2004) D. Lucrédio, E. S. Almeida, A. F. Prado, **A Survey on Software Components Search and Retrieval**, *30th IEEE EUROMICRO Conference on Software Engineering and Advanced Applications* (EUROMICRO-SEAA), Component-Based Software Engineering Track, Rennes, France, August/September, 2004, pp. 152-159.

(Lucrédio et al., 2004a) D. Lucrédio, C. P. Bianchini, A. F. Prado, L. C. Trevelin, E. S. Almeida, **ORION - A Component-Based Software Engineering Environment,** *Journal of Object Technology* (JOT), Vol. 03, No. 04, November, 2004, pp. 51-74.

(Martin, 1994) R. Martin, **OO Design Quality Metrics: An Analysis of Dependencies**, Consulted in June 05, 2006, Available in http://www.objectmentor.com/resources/articles/oodmetrc.pdf, October, 1994, pp.08.

(Mascena et al., 2005) J. C. C. P. Mascena, E. S. Almeida, S. R. L. Meira, **A Comparative Study on Software Reuse Metrics and Economic Models from a Traceability Perspective**, *IEEE International Conference on Information Reuse and Integration* (IRI), Las Vegas, USA, August, 2005, pp. 72-77.

(Mascena, 2006) J. C. C. P. Mascena, **ADMIRE: Asset Development Metric-based Integrated Reuse Environment**, M.Sc. Dissertation, Federal University of Pernambuco, May, 2006, pp. 126.

(Mascena et al., 2006a) J. C. C. P. Mascena, V. C. Garcia, E. S. Almeida, S. R. L. Meira, **A Comparative Study on Software Reuse Metrics and Economic Models from a Traceability Perspective**, *5th ACM International Conference on Generative Programming* (GPCE), Portland, USA, October, 2006.

(Matinlassi, 2004) M. Matinlassi, **Comparison of Software Product Line Architecture Design Methods: COPA, FAST, FORM, KobrA and QUADA**, *26th International Conference on Software Engineering* (ICSE), Edinburgh, Scotland, May, 2004, pp. 127-136.

(McCabe, 1976) T. J. McCabe, **A Complexity Measure**, *IEEE Transactions on Software Engineering*, Vol. 02, No. 04, December, 1976, pp. 308-320.

(McConnell, 1998) S. McConnell, **The Power of Process**, *IEEE Computer*, Vol. 31, No. 05, May, 1998, pp. 100-102.

(McGibbon et al., 2001) B. McGibbon, **Status of CBSE in Europe**, in Component-Based Software Engineering: Putting the Pieces Together, G. T. Heineman, B. Councill, Addison-Wesley, 2001, pp. 818.

(McGregor et al., 2002) J. D. McGregor, L. M. Northrop, S. Jarrad, K. Pohl, **Initiating Software Product Lines**, *IEEE Software*, Vol. 19, No. 04, July/August, 2002, pp. 24-27.

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

(Mei et al., 2003) H. Mei, W. Zhang, F. Gu, **A Feature Oriented Approach to Modeling and Reusing Requirements of Software Product Lines**, *27th IEEE International Computer Software and Applications Conference* (COMPSAC), Dallas, Texas, USA, November, 2003, pp. 250-256.

(Mernik et al., 2005) M. Mernik, J. Heering, A. M. Sloane, **When and How to Develop Domain-Specific Languages**, *ACM Computing Surveys*, Vol. 37, No. 04, December, 2005, pp. 316-344.

(Meyer, 1996) B. Meyer, **The Reusability Challenge,** *IEEE Computer*, Vol. 29, No. 02, February, 1996, pp. 76-78.

(Meyer et al., 1998) B. Meyer, C. Mingins, H. Schmidt, **Providing Trusted Components to the Industry**, *IEEE Computer*, Vol. 31, No. 05, May, 1998, pp. 104-105.

(Meyer & Mingins, 1999) B. Meyer, C. Mingins, **Component-based development: from Buzz to Spark**, *IEEE Computer*, Vol. 32, No. 07, July, 1999, pp. 35-36.

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

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

(Mili et al., 1999) A. Mili, S. Yacoub, E. Addy, H. Mili, **Toward an Engineering Discipline of Software Reuse**, *IEEE Software*, Vol. 16, No. 05, September/October, 1999, pp. 22-31.

(Mili et al., 2002) H. Mili, A. Mili, S. Yacoub, E. Addy, **Reuse-Based Software Engineering**, Willey, 2002, pp. 636.

(Mollaghasemi & Pet-Edwards, 1997) M. Mollaghasemi, J. Pet-Edwards, **Making Multiple-Objective Decisions**, Institute of Electrical & Electronics Engineer, 1997, pp. 91.

(Moon et al., 2005) M. Moon, K. Yeom, **An Approach to Developing Domain Requirements as a Core Asset Based on Commonality and Variability Analysis in a Product Lines**, *IEEE Transactions on Software Engineering*, Vol. 31, No. 07, July, 2005, pp.551-569.

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

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

(Muller, 1995) J. K. Muller, **Integrating Architectural Design into the Development Process**, *International Symposium and Workshop on Systems Engineering of Computer-Based Systems*, Tucson, USA, March, 1995, pp. 114-121.

(Muthig & Atkinson, 2002) D. Muthig, C. Atkinson, **Model-Driven Product Line Architectures**, 2nd *International Conference on Software Product Lines* (SPLC), San Diego, USA, August, 2002, pp. 110-129.

(Neighbors, 1980) J. M. Neighbors, **Software Construction Using Components**, Ph.D. Thesis, University of California, Irvine, Department of Information and Computer Science, USA, April, 1980, pp.217.

(Neighbors, 1989) J. M. Neighbors, **Draco: A Method for Engineering Reusable Software Systems**, in Software Reusability Volume I: Concepts and Models, T. Biggerstaff, A. Perlis, 1989, pp. 425.

(Northrop, 2002) L. M. Northrop, **SEI's Software Product Line Tenets**, *IEEE Software*, Vol. 19, No. 04, July/August, 2002, pp. 32-40.

(OMG, 2005) **Unified Modeling Language: Superstructure**, *Formal Specification*, July, 2005, pp. 710.

(OMG, 2005a) **Software Process Engineering Metamodel Specification**, *Formal Specification*, January, 2005, pp. 99.

(Ommering, 2005) R. C. van Ommering, **Software Reuse in Product Populations**, *IEEE Transactions on Software Engineering*, Vol. 31, No. 07, July, 2005, pp. 537-550.

(OSGi, 2005) **OSGi Service Platform Core Specification**, *Core Specification*, Release 4, August, 2005, pp. 276.

(Osterweil, 1987) L. Osterweil, **Software processes are software too,** *9th International Conference on Software Engineering* (ICSE), Monterey, California, USA, March/April, 1987, pp. 02-13.

(Oxford, 2000) Oxford University, **Oxford Advanced Learner's Dictionary**, Oxford University Press, 2000, pp. 1540.

(Paller, 2006) G. Paller, **Building Reflective Mobile Middleware Framework on Top of the OSGi Platform**, *9th International Conference on Software Reuse* (ICSR), Torino, Italy, June, 2006, pp. 354-367.

(Papazoglou & Georgakopoulos, 2003) M. P. Papazoglou, D. Georgakopoulos, **Service-Oriented Computing**, *Communications of the ACM*, Vol. 46, No. 10, October, 2003, pp. 25-28.

(Parnas, 1972) D. L. Parnas, **On the Criteria to be Used in Decomposing Systems into Modules**, *Communications of the ACM*, Vol. 15, No. 12, December, 1972, pp. 1053-1058.

(Parnas, 1976) D. L. Parnas, **On the Design and Development of Program Families**, *IEEE Transactions on Software Engineering*, Vol. 02, No. 01, March, 1976, pp. 01-09.

(Perry & Wolf, 1992) D. E. Perry, A. L. Wolf, **Foundations for the Study of Software Architecture**, *ACM SIGSOFT Software Engineering Notes*, Vol. 17, No. 04, October, 1992, pp. 40-52.

(Pfleeger, 1996) S. L. Pfleeger, **Measuring Reuse: A Cautionary Tale**, *IEEE Software*, Vol. 13, No. 04, July, 1996, pp. 118-127.

(Phoha, 1997) V. Phoha, **A Standard for Software Documentation**, *IEEE Computer*, Vol. 30, No. 10, October, 1997, pp. 97-98.

(Pohl et al., 2005) K. Pohl, G. Bockle, F. van der Linden, **Software Product Line Engineering: Foundations, Principles, and Techniques**, Springer, 2005, pp. 467.

(Poulin, 1997) J. S. Poulin, **Measuring Software Reuse**, Addison-Wesley, 1997, pp. 195.

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

(Poulin, 2006) J. S. Poulin, **The Business Case for Software Reuse: Reuse Metrics, Economic Models, Organizational Issues, and Case Studies**, *Tutorial Notes*, Torino, Italy, June, 2006.

(Pressman, 2005) R. S. Pressman, **Software Engineering: A Practitioner's Approach,** McGraw-Hill, 2005, pp. 880.

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

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

(Rada & Moore, 1997) R. Rada, J. Moore, **Standardizing reuse**, *Communications of the ACM*, Vol. 40, No. 03, March, 1997, pp. 19-23.

(Ran, 1999) A. Ran, **Software isn't built from Lego blocks**, *Symposium on Software Reusability* (SSR), Los Angeles, USA, May, 1999, pp. 164-169.

(Ran, 2000) A. Ran, **ARES Conceptual Framework for Software Architecture**, in Software Architecture for Product Family: Principles and Practice, M. Jazayeri, A. Ran, F. van der Linden, Addison-Wesley, 2000, pp. 257.

(Ravichandran & Rothenberger, 2003) T. Ravichandran, M. A. Rothenberger, **Software reuse strategies and component markets**,

Communications of the ACM, Vol. 46, No. 08, August, 2003, pp. 109-114.

(Reifer, 1997) D. J. Reifer, **Practical Software Reuse**, Addison-Wesley, 1997, pp. 374.

(Repenning et al., 2001) A. Repenning, A. Ioannidou, M. Payton, W. Ye, J. Roschelle, **Using Components for Rapid Distributed Software Development**, *IEEE Software*, Vol. 18, No. 02, March/April, 2001, pp.38-45.

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

(Rine, 1997a) D. C. Rine, **Supporting Reuse with Object Technology**, *IEEE Computer*, Vol. 30, No. 10, October, 1997, pp. 43-45.

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

(Ross, 1977) D. T. Ross, **Structured Analysis (SA): A Language for Communicating Ideas**, *IEEE Transaction on Software Engineering*, Vol. 03, No. 01, January, 1977, pp. 16-34.

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

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

(Sarang et al., 2001) P. G. Sarang, K. Gabhart, A. Tost, T. McAllister, R. Adatia, M. Juric, T. Osborne, F. Arni, J. Lott, V. Nagarajan, C. A. Berry, D. O'Connor, J. Griffin, A. Mulder, D. Young, **Professional EJB**, Wrox Press, July, 2001, pp. 1200.

(Schneider & Han, 2004) J. G. Schneider, J. Han, **Components – the Past, the Present, and the Future**, *9th International Workshop on Component-Oriented Programming* (WCOP), Oslo, Norway, June, 2004, pp. 08.

(Schmid et al., 2001) K. Schmid, S. Thiel, J. Bosch, S. Johnsson, M. Jaring, B. Thomé, **Scoping**, Eureka ∑! 2023 Programme, ITEA project, June, 2001, pp. 67.

(Schmidt, 2006) D. C. Schmidt, **Model-Driven Engineering**, *IEEE Computer*, Vol. 39, No. 02, February, 2006, pp. 25-31.

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

(Shaw & Garlan, 1996) M. Shaw, D. Garlan, **Software Architecture: Perspective on an Emerging Discipline,** Prentice Hall, pp. 242.

(Sherif et al., 2006) K. Sherif, R. Appan, Z. Lin, **Resources and incentives for the adoption of systematic software reuse**, *International Journal of Information Management*, Vol. 26, No. 01, February, 2006, pp. 70–80.

(Silva & Werner, 1996) M. F. Silva, C. M. L. Werner, **Packaging Reusable Components using Patterns and Hypermedia**, *4th International Conference on Software Reuse*, Orlando, Florida, ,USA, April, 1996, pp. 146-155.

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

(Smaragdakis & Batory, 1998) Y. Smaragdakis, D. Batory, **Implementing Reusable Object-Oriented Components**, *5th International Conference on Software Reuse* (ICSR), Victoria, Canada, June, 1998, pp. 36-45.

(Sommerville, 2006) I. Sommerville, **Software Engineering,** Addison-Wesley, pp. 840.

(Sparling, 2000) M. Sparling, **Lessons Learned Through six years of Component-Based Development**, *Communications of the ACM*, Vol. 43, No. 10, October, 2000, pp. 47-53.

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

(STARS, 1996) **Software Technology for Adaptable, Reliable Systems (STARS) Program, DAGAR: A Process for Domain Architecture Definition and Asset Implementation**, *Technical Report*, April, 1996, pp. 33.

(Svahnberg et al., 2001) M. Svahnberg, J. van Gurp, J. Bosch, **On the Notion of Variabilities in Software Product Lines**, *Working IEEE/IFIP Conference on Software Architecture* (WICSA), Amsterdam, Netherlands, August, 2001, pp. 45-54.

(Sugumaran et al., 1999) V. Sugumaran, M. Tanniru, V. C. Storey, **Identifying Software Components from Process Requirements using Domain Model and Objects Libraries**, *20th ACM International Conference on Information System*, Charlotte, North Carolina, USA, December, 1999, pp. 65-81.

(Szyperski, 2002) C. Szyperski, **Component Software: Beyond Object-Oriented Programming**, Addison-Wesley, 2002, pp. 588.

(Taulavuori et al., 2004) A. Taulavuori, E. Niemela, P. Kallio, **Component documentation—a key issue in software product lines**, *Journal Information and Software Technology*, Vol. 46, No. 08, June, 2004, pp. 535–546.

(Traas & van Hillegersberg, 2000) V. Traas, J. van Hillegersberg, **The software component market on the internet current status and conditions for growth**, *ACM SIGSOFT Software Engineering Notes*, Vol. 25, No. 01, January, 2000, pp. 114-117.

(Tracz, 1995) W. Tracz, **DSSA (Domain-Specific Software Architecture) Pedagogical Example**, *ACM SIGSOFT Software Engineering Notes*, Vol. 20, No. 03, July, 1995, pp. 49-62.

(Vanderlei et al., 2007) T. A. Vanderlei, F. A. Durão, A. C. Martins, V. C. Garcia, E. S. Almeida, S. R. L. Meira, **A Classification Mechanism for Search and Retrieval Software Components**, *22nd Annual ACM Symposium on Applied Computing* (SAC), Information Retrieval Track, Seul, Korea, March, 2007.

(VanDoren, 1997) E. VanDoren, **Maintainability Index Technique for Measuring Program Maintainability**, Software Engineering Institute (SEI), Available in http://www.sei.cmu.edu/str/descriptions/mitmpm.html, Consulted in September 21, 2006.

(Villela, 2000) R. M. M. B. Villela, **Search and Recovery of Components in Software Reuse Environments** (*in portuguese*), Ph.D. Thesis, Federal University of Rio de Janeiro, December, 2000, pp. 264.

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

(Voas, 1998) J. Voas, **Maintaining Component-Based Systems**, *IEEE Software*, Vol. 15, No. 04, July/August, 1998, pp. 22-27.

(Voth, 2004) D. Voth, **Packaging Reusable Software Assets**, *IEEE Software*, Vol. 21, No. 03, May/June, 2004, pp. 107-110.

(Warmer & Kleppe, 1999) J. Warmer, A. Kleppe, **The Object Constraint Language – Precise Modeling with UML**, Addison-Wesley, 1999.

(Weiss & Lai, 1999) D. M. Weiss, C. T. R. Lai, **Software Product-Line Engineering: A Family-Based Software Development Process**, Addison-Wesley, 1999, pp. 426.

(White & Gallaher, 2002) W. J. White, M. P. Gallaher, **Benefits and Costs of ATP Investments in Component-Based Software**, National Institute of Standards and Technology, *Technical Report*, November, 2002, pp. 141.

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

(Wohlin et al., 2000) C. Wohlin, P. Runeson, M. Host, M. C. Ohlsson, B. Regnell, A. Wesslén, **Experimentation in Software Engineering: An Introduction**, Kluwer Academic Publishers, 2000, pp. 204.

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

Appendix A. Recommended References

"Don't believe everything you read"

Barry Boehm (2006)
Professor of Software Engineering, University of Southern California

---

Component-Based Development (CBD)

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

In this revolutionary paper, McIlroy presents his thesis saying: "the software industry is weakly founded and that one aspect of this weakness is the absence of a software components sub industry". McIlroy proposes an investigation in mass-production techniques in software, according to some ideas from the construction industry, and presents a proposal of how a component factory should function.

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

In this paper, Cox presents an important consideration about the software industrial revolution making an analogy between the hardware engineering and software engineering based on reusable assets as software components.

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

In this important paper about experiences with CBD, a nationwide CDB promotion project is discussed, with explanation and emphasis of the insights and six lessons learned from the project. The paper could be better explored if was presented quantitative and qualitative data about the project.

Component-Based Development (CBD) Books

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

Sametinger presented the first book about software components. The book covers important aspects related to it such as adaptation, composition, repository systems, etc. This book was very important to present the first ideas about the theme in a concise way.

(D'Souza & Wills, 1998) D. F. D'Souza, A. C. Wills, **Objects, Components and Frameworks with UML: The Catalysis Approach**, Addison-Wesley, 1998, pp. 816.

In this book, D'Souza and Wills present the Catalysis approach for CBD. The book can be seen as the first one related to a CBD method. However, the book is too extensive and does not discuss the ideas in a systematic way for a method. Several ideas are discussed, but the reader does not have a common direction.

(Cheesman & Daniels, 2000) J. Cheesman, J. Daniels, **UML Component A Simple Process for Specifying Component-Based Software**, Addison-Wesley, 2000, pp. 208.

In this book, Cheesman and Daniels present the UML Component method. The method has been widely used, especially, in the academic world due to simplicity reasons. UML Component instead of Catalysis defines a simple and well-defined method to develop software components. However, some important activities are not discussed such as domain analysis or domain architecture.

(Heineman & Councill, 2001) G. T. Heineman, W. T. Councill, **Component-Based Software Engineering**, Addison-Wesley, 2001, pp. 818.

Heineman and Council organized a complete handbook about software components discussing several aspects related to it with different point of views based on experts in the theme. The book discusses also other aspects associated to CBD such as product lines, software architecture besides presenting future directions in the area.

(Szyperski, 2002) C. Szyperski, **Component Software: Beyond Object-Oriented Programming**, Addison-Wesley, 2002, pp. 588.

Szyperski the foremost researcher in the CBD area discusses widely the theme since its roots, reflections with object-oriented programming until its future. The book is seen as classic by researchers and practitioners working on the subject.

Domain Analysis (DA)

(Arango, 1989) G. Arango, **Domain analysis: from art form to engineering discipline**, *5th International Workshop on Software specification and design*, Pittsburgh, Pennsylvania, USA, May, 1989, pp. 152-159.

In this paper, Arango presents an overview on domain analysis and discusses a possible way to try to formalize the process. The model is general, but it can be useful to design new domain analysis methods.

(Kang et al., 1990) K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, A. S. Peterson, **Feature-Oriented Domain Analysis (FODA) Feasibility Study**, Software Engineering Institute (SEI), *Technical Report*, November, 1990, pp. 161.

Kang et al. propose the first systematic approach to perform domain analysis and define also the main concept used in the area: the feature model. It can be seen as one of the main contributions in the domain analysis field and it is used even currently, seventeen years after their work.

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

In this classic paper, Rubén Prieto-Díaz presents a brief introduction for domain analysis area, discussing points such as experiences, basic definitions, research, and the whole process.

Empirical Studies in Software Reuse

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

In this paper, Rine briefly presents a 1995 survey conducted to investigate what are the success factors for software reuse that are applicable across all application domains. The study indicates that the practices associated with top predictors of software reuse capability match the successful practices used in hardware manufacturing.

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

This paper, maybe the more detailed practical study about software reuse available in the literature, presents and discusses some of the key factors in adopting or running a company-wide software reuse program. The key factors are derived from empirical evidence associated to the analysis of twenty-four projects performed from 1994 to 1997.

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

This research investigates 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. Rothenberger et al. use survey data collected from 71 software development groups to empirically develop a set of six dimensions that describe the practices employed in reuse programs.

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

Selby presents a 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 were analyzed.

Experimental Software Engineering

(Basili et al., 1985) V. R. Basili, R. W. Selby, D. H. Hutchins, **Experimentation in Software Engineering**, *IEEE Transactions on Software Engineering*, Vol. 12, No. 07, July, 1986, pp. 733-743.

In this conceptual paper on the theme, Basili et al. present a framework for analyzing most of the experimental work that has been performed in software engineering over the years, discussing their experiments, results, and impact.

(Basili, 1996) V. R. Basili, **The Role of Experimentation in Software Engineering: Past, Present, and Future**, *18th International Conference on Software Engineering* (ICSE), Berlin, Germany, March, 1996, pp. 442-449.

In this classic paper on the area, Basili presents and discusses the experimental paradigm and outlines a classification scheme for characterizing such experiments.

Experimental Software Engineering Books

(Wohlin et al., 2000) C. Wohlin, P. Runeson, M. Host, M. C. Ohlsson, B. Regnell, A. Wesslén, **Experimentation in Software Engineering: An Introduction**, Kluwer Academic Publishers, 2000, pp. 204.

Wohlin et al. present the first and foremost book about experimental software engineering. The book covers the main steps to perform an experimental study since its definition until the analysis and interpretation. The book is essential for anyone working on this subject.

(Boehm et al., 2005) B. W. Boehm, H. D. Rombach, M. V. Zelkowitz, **Foundations of Empirical Software Engineering: The Legacy of Victor R. Basili**, Springer, 2005, pp. 431.

In this book, Boehm and his colleagues organize the main contributions of Victor Basili in the experimental software engineering area. The book discusses its main papers on the point of view of specialists such as Weiss, Zelkowitz, Rombach and Boehm.

Libraries and Repository Systems

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

In this classic paper, Prieto and Freeman present a partial solution for the component retrieval problem. The work proposes a faceted classification scheme based on reusability-related attributes and a selection mechanism as a solution to the software reuse problem. This paper has influenced strongly the research in the component retrieval area.

Metrics

(McCabe, 1976) T. J. McCabe, **A Complexity Measure**, *IEEE Transactions on Software Engineering*, Vol. 02, No. 04, December, 1976, pp. 308-320.

In this paper, McCabe proposes a graph-theoretic complexity measure and illustrates how it can be used to manage and control program complexity.

Patterns

(Alexander et al., 1977) C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. F. King, S. Angel, **A Pattern Language: Towns, Buildings, Construction**, Oxford University Press, 1977, pp. 1216.

In this book, Alexander et al. describe a pattern language used to build towns and buildings. The book was the first one related to the theme and it influenced strongly the software development area with its ideas and organization for describing the patterns.

(Gamma et al., 1995) E. Gamma, R. Helm, R. Johnson, J. Vlissides, **Design Patterns: Elements of Reusable Object-Oriented Software**, Addison-Wesley, 1995, pp. 395.

In this classical book, Gamma and his colleagues present the main book on software patterns. They describe their experiences in building object-oriented software discussing recurring problems and a set of elegant solutions.

Software Architecture

(Parnas, 1972) D. L. Parnas, **On the Criteria to be Used in Decomposing Systems into Modules**, *Communications of the ACM*, Vol. 15, No. 12, December, 1972, pp. 1053-1058.

In this classic paper, Parnas presents the idea of decomposing systems in modules and defines concepts such as coupling, information hiding, etc. These concepts are currently the foundations for software development.

(Perry & Wolf, 1992) D. E. Perry, A. L. Wolf, **Foundations for the Study of Software Architecture**, *ACM SIGSOFT Software Engineering Notes*, Vol. 17, No. 04, October, 1992, pp. 40-52.

In this seminal paper on software architecture, Perry and Wolf discuss the main foundations for the field, the main definitions, and efforts in the area.

(Kazman et al., 1994) R. Kazman, L. J. Bass, M. Webb, G. D. Abowd, **SAAM: A Method for Analyzing the Properties of Software Architecture**, *16th International Conference on Software Engineering* (ICSE), Sorrento, Italy, May, 1994, pp. 81-90.

Kazman et al., present an important topic on software architecture: how to evaluate it in a systematic way. They propose and discuss the SAAM method and its utilization in the industry. The paper was a strong contribution to define new methods and approaches in the area.

(Kruchten, 1995) P. Kruchten, **The 4 + 1 View Model of Architecture**, *IEEE Software*, Vol. 12, No. 06, November, 1995, pp. 45-50.

In this work, Kruchten propose the 4 + 1 model of architecture. This model influenced strongly the software development industry and it was incorporated in industrial processes such as the Rational Unified Process.

(Tracz, 1995) W. Tracz, **DSSA (Domain-Specific Software Architecture) Pedagogical Example**, *ACM SIGSOFT Software Engineering Notes*, Vol. 20, No. 03, July, 1995, pp. 49-62.

In this paper, Tracz discusses the domain-specific software architecture field through a pedagogical example. It can be seen as the first effort to systematize the process of designing DSSA.

(Kruchten et al., 2006) P. Kruchten, H. Obbink, J. Stafford, **The Past, Present, and Future of Software Architecture**, *IEEE Software*, Vol. 23, No. 02, March-April, 2006, pp. 22-30.

In this paper, Kruchten et al. discuss the software architecture area, its foundations, roots, main efforts, results and its future. The paper can be seen as the first step to discover the area, the main conferences, researchers, and efforts besides defining a strong base for it.

Software Architecture Books

(Shaw & Garlan, 1996) M. Shaw, D. Garlan, **Software Architecture: Perspective on an Emerging Discipline,** Prentice Hall, pp. 242.

This book is the first contribution for the field in this sense. The book is considered a classical on the field discussing the main topics on software architecture on the optic of two important researchers in the area.

(Bass et al., 2003) L. Bass, P. Clements, R. Kazman, **Software Architecture in Practice**, 2nd Edition, Addison-Wesley, 2003, pp. 560.

This book can be seen as a complement for the previous one. It was written based on the practical experience on the field at Software Engineering Institute showing real case studies and the main foundations for the area.

Software Development Problems

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

This paper presents the importance of Design by Contract (DBC) in the software development based on the lessons of Ariane. 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.

Software Documentation

(Lethbridge et al., 2003) T. C. Lethbridge, J. Singer, A. Forward, **How Software Engineers Use Documentation: The State of the Practice**, *IEEE Software*, Vol. 20, No. 06, November/December, 2003, pp. 35-39.

In this paper, Lethbridge et al. discuss three industrial studies in order to identify the patterns by which software engineers use and update documentation.

Software Engineering

(Pressman, 2005) R. S. Pressman, **Software Engineering: A Practitioner's Approach,** McGraw-Hill, 2005, pp. 880.

In this classic book on software engineering, Pressman discusses widely the topic involving the main disciplines and future directions on the field. This book in conjunction with Sommerville (2006) is essential for anyone working with software engineering.

(Boehm, 2006) B. W. Boehm, **A View of 20th and 21st Century Software Engineering,** *28th International Conference on Software Engineering* (ICSE), Shanghai, China, May, 2006, pp. 12-29.

In this paper, Boehm presents his reflection on the software engineering field discussing its roots, the main contributions, silver bullets and directions for future research.

(Sommerville, 2006) I. Sommerville, **Software Engineering,** Addison-Wesley, pp. 840.

This book is as important as the Pressman's book presenting an additional vision on the field.

Software Process

(Osterweil, 1987) L. Osterweil, **Software processes are software too,** *9th International Conference on Software Engineering* (ICSE), Monterey, California, USA, 1987, March-April, pp. 02-13.

In this important paper, Osterweil defined a polemic point of view discussing that software processes should be seen as software too. The paper was a considerable effort to discuss new research on the field.

(Boehm, 1988) B. W. Boehm, **A Spiral Model of Software Development and Enhancement**, *IEEE Computer*, Vol. 21, No. 05, May, 1988, pp. 61-72.

In this classic paper, Boehm proposes the spiral model for software development. His ideas on this work are used until today by the software industry and presented an additional direction for software development.

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

In this book, the main specialists in software development propose and discuss the Unified Software Development Process which influenced strongly the software development community.

Software Product Lines

(Parnas, 1976) D. L. Parnas, **On the Design and Development of Program Families**, *IEEE Transactions on Software Engineering*, Vol. 02, No. 01, March, 1976, pp. 01-09.

In this paper, Parnas presented the initial and main ideas on the development of program families. The paper was the first attempt to change the vision of developing a set of systems in the same domain as a family instead of working in an isolated way.

(Svahnberg et al., 2001) M. Svahnberg, J. van Gurp, J. Bosch, **On the Notion of Variabilities in Software Product Lines**, *Working IEEE/IFIP Conference on Software Architecture* (WICSA), Amsterdam, Netherlands, August, 2001, pp. 45-54.

In this paper, Svahnberg et al. perform a wide discussion on variabilities in the context of software product lines approaching its definition, taxonomies, and variabilities types.

Software Reuse

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

In this paper, Basili et al. present a comprehensive framework for reuse, based on a number of assumptions regarding software development in general and reuse in particular. Instant of assumptions, are discussed the reuse model and the reuse-oriented TAME environment model, which supports the integration of reuse into software development. Although interesting, the paper could present more details about the reuse-oriented software development model, as well as the activities, artifacts and results.

(Ommering, 2005) R. C. van Ommering, **Software Reuse in Product Populations**, *IEEE Transactions on Software Engineering*, Vol. 31, No. 07, July, 2005, pp. 537-550.

In this paper, Ommering present and discuss the efforts performed with software product lines at Phillips in the consumer electronic domain.

Software Reuse Books

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

Jacobson et al. published the first 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.

(Poulin, 1997) J. S. Poulin, **Measuring Software Reuse**, Addison-Wesley, 1997, pp. 195.

In this classic book, Poulin presents important contributions to research related to software reuse, as for example, How to measure the benefit of software reuse? What the relative costs of developing for and with reuse? For this book, we agree with Will Tracz when he said: "This is the best book on software reuse metrics available".

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

Clements & Northrop present an important conceptual book on software product lines. The book based on the Software Engineering Institute's experience on the field discuss the technical and non-technical aspects related to software product lines through a framework defined at SEI.

Software Reuse Company Reports

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

In 1991, the Reuse Technology Support Center was established to coordinate and manage the reuse activities within IBM. 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 in software reuse. In this paper, Bauer describes how the parts center evolved and the results of this work.

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

In this paper, Albert Endres a senior IBM' worker presents his experiences based on more 20 years at the IBM. Two aspects are discussed: formal methods in development, and software reuse.

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

In this paper, Rebecca Joos describes her experience on defining and introducing software reuse at Motorola. This process was composed of three phases: Grass-Roots Beginning, Senior-Management Involvement and Future Markets and Tools. These phases are presented and discussed in detail. The paper is one more about the software reuse experience in companies, showing that without an educational base, software reuse does not work.

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

In this paper about software reuse experience at HP, M. Griss presents a little more of his experience at company; discusses some myths (five) related to software reuse; and, briefly, describes an incremental model for reuse introduction.

Software Reuse Processes

(Neighbors, 1980) J. M. Neighbors, **Software Construction Using Components**, PhD Thesis, University of California, Irvine, Department of Information and Computer Science, USA, April, 1980, pp.217.

This work present the first domain engineering approach based on new concepts such as DSLs, generative programming and a transformation machine. The work was very important for the field development.

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

This guidebook describes the Organization Domain Modeling (ODM) in details discussing its three phases: Plan Domain, Model Domain, and Engineer Asset Base. The guidebook also presents some guidelines in order to adopt ODM in organizations, however, quantitative and qualitative data does not discussed, as well as details about the projects that used the method.

(Kang et al., 1998) K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, M. Huh, **FORM: A Feature-Oriented Reuse Method with domain-specific reference architectures**, *Annals of Software Engineering Notes*, Vol. 05, No. 00, Janeiro, 1998, pp. 143-168.

In this paper, Kang et al. present the FORM (Feature-Oriented Reuse Method). FORM extends FODA with directions in domain design and implementation and some considerations related to application development reusing the artifacts developed.

(Weiss & Lai, 1999) D. M. Weiss, C. T. R. Lai, **Software Product-Line Engineering: A Family-Based Software Development Process**, Addison-Wesley, 1999, pp. 426.

Weiss & Lai present the first book related to software product lines. In this book, they propose the FAST method and discuss the topic through their experience in industry.

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

In this paper, Atkinson et al. present the KobrA approach. The approach represents a synthesis of several advanced software engineering technologies, including product line, component-based development, frameworks, and process modeling, in order to support the software product line development.

Software Reuse Surveys

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

In this exhaustive survey, Charles Krueger presents the different approaches to software reuse found in the research literature (eight). Krueger 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.

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

In this survey, Mili et al. present an important overview about software reuse. In opposite Krueger, that described the different approaches to software reuse, Mili reported other relevant aspects of software reuse, including: the new process models; some economic models and software metrics; object-oriented programming and its implication on software reuse; the component retrieval problem; and aspects such as composition, and adaptation of software components.

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

Mili et al. present and discuss the main approaches related to software reuse libraries. This paper is obligatory reading for researchers working with libraries and repository systems.

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

In this paper, Frakes & Kang present a summary on the software reuse research discussing unsolved problems, based on the Eighth International Conference on Software Reuse (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.

Success and Failure Factors in Software Reuse

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

In this short paper, Card and Comer present the two fundamental mistakes that contribute to failure in software reuse. Additionally, it is discussed a cost model for software reuse and the four important factors for an effective software reuse program (Training, Incentives, Measurement and Management).

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

In this paper, Frakes & Isoda present and discuss six success factors of systematic reuse (Management, Measurement, Legal issues, Economics, Design for reuse and Libraries). Additionally, is presented an interesting software reuse reading list.

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

In this important paper, Frakes and Fox analyze sixteen questions about software reuse using survey data collected from organizations in the U.S and Europe.

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

In this paper, Glass presents and discusses the question that software reuse has problems in the practice. Glass considers that the big problem with software reuse is that there are not many software components that can be reused. At the end, is briefly presented and discussed some important factors in software reuse.

Appendix B. References Distribution

This appendix presents the references distribution used in this thesis. Figure B.1 shows the relationship between the references source and its occurrence. On the other hand, Figure B.2 shows a different view analyzing the references year and its occurrence. It can be useful to show the investigated sources during the thesis, their distribution and how old or new are them.

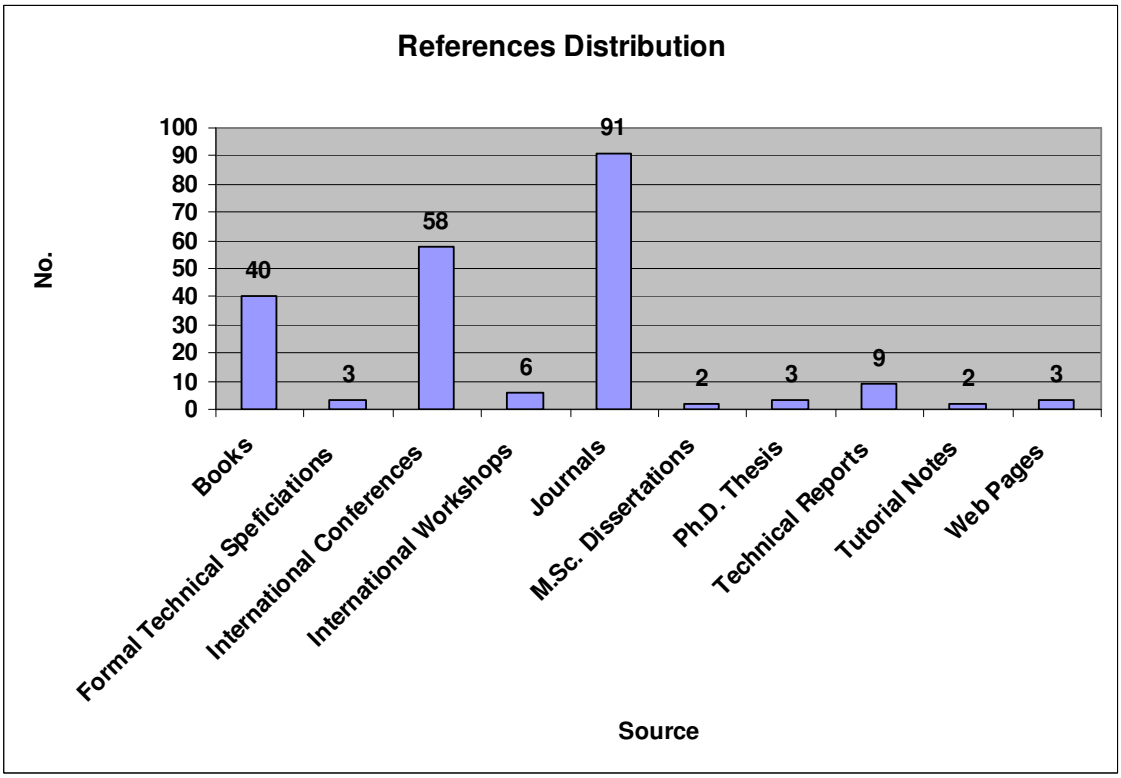

Figure B.1. Distribution of References by Source.

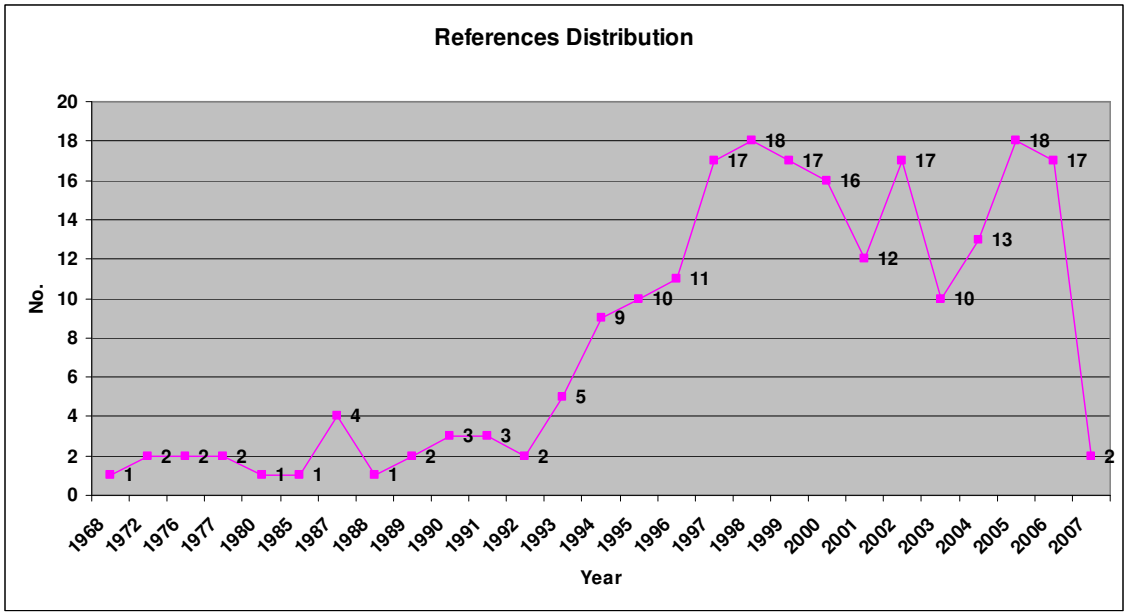

Figure B.2. Distribution of References by Year.

Appendix C. Domain Scope Template

In this template, the features, sub-features, applications (existing, future, and potential), and characterization and benefit functions are represented composing the scope definition.

ID Feat.	Feat. Name	ID SubFeat.	SubFeat. Name	ID SubFeat.	SubFeat. Name	Application 1				Application 2				Benefit			
						Req	Sim	Eff (c,a)	Eff (c)	Req	Sim	Eff (c,a)	Eff (c)	P(c)	D(c)	EA(c)	E(c)
[F 01]														0	0	0	0
														0	0	0	0
														0	0	0	0
														0	0	0	0
														0	0	0	0
														0	0	0	0
														0	0	0	0
														0	0	0	0
														0	0	0	0
														0	0	0	0
														0	0	0	0
														0	0	0	0
														0	0	0	0
														0	0	0	0
[F 02]														0	0	0	0
														0	0	0	0
														0	0	0	0
														0	0	0	0
														0	0	0	0
														0	0	0	0
														0	0	0	0
														0	0	0	0
														0	0	0	0

Appendix D. Domain Documentation Template

In this template, the domain analysis step is described in details, especially, the planning, the applications, the features and the domain.

Planning

This section describes the preparation process for domain analysis.

Stakeholder analysis	
Objectives definition	
Constraint definition	
Market analysis	
Data collection	

Applications

This section describes the existing, future and potential applications in the domain.

Existing Applications	
Future Applications	
Potential Applications	

Feature Model

This section shows the domain feature model.

Domain Documentation

This section describes the domain documentation.

Domain's Name
Description
Defining rules
Exemplar system selection
Documentation

Domain Context
Domain genealogy

Feature Documentation

This section describes each feature in details.

Feature's Name
Semantic Description
Rationale
Stakeholders and client programs
Exemplar Applications
Constraints
Variation Point

() open **()** closed
Priority

Appendix E. Component Grouping Template

In this template, the assets produced during the *group component* sub-activity are documented.

Use Cases Clustering

This section documents the functional dependency and the cluster step among use cases.

Functional Dependency Matrix

	Weight (W)
M_{11} = Sub-system	
M_{12} = Actor	
M_{13} = Shared Data	
M_{14} = Relationships	
FD = Functional Dependency	

		[UC01]		[UC02]		...	
	Measure Criteria	M	(M * W)	M	(M * W)	M	(M * W)
[UC01]	M_{11}						
	M_{12}						
	M_{13}						
	M_{14}						
	FD						
[UC02]	M_{11}						
	M_{12}						
	M_{13}						
	M_{14}						
	FD						

....	M_{11}						
	M_{12}						
	M_{13}						
	M_{14}						
	FD						

Allocating Classes to Components

This section presents the classes allocated to components and the relationships and weight used to solve conflict in the *allocate classes to components* step.

Classes	Components

Relationship	Weight
Dependency	
Aggregation	
Association	
Inheritance	
Composition	

Components

This section presents the internal structure of each component identified with its related classes.

Components	Classes

Appendix F. Component Specification Template (CST)

This template is responsible for representing the component specification defined in Domain Design step. For each component, it is presents its description, use cases, features, workflow, class diagrams, interfaces, packaging, and quality attributes.

Components

[CP 01] <Component Name>

Name:	
Description:	
Use Cases:	
Features:	

Workflow

This section describes the component workflow.

Classes

This section presents the component internal structure with its classes and relationships.

Business Interface

This section shows the component business interface.

Provided Interfaces

This section presents the component provided interfaces and its services offered.

Required Interfaces

This section presents the component required interfaces.

Component Packaging

This section describes the internal component packaging.

Quality Attributes

This section presents the component quality attributes.

Appendix G. Domain Architecture Template

This template is responsible for documenting the domain architecture defined in Domain Design step presenting its goals, modules, quality attributes, views, and component diagrams.

Architecture Goals and Constraints

This section describes the goals and architectural constraints.

Modules

This section presents the name and description of each module in the architecture.

Architecture Views

This section describes the architectural views for the designed architecture.

Module View

This section presents the module view of the domain architecture.

[Module Name]

Name:	
Description:	
Use Cases:	
Features:	

Quality Attributes

This section describes the quality attributes for each module.

[Sub-Module Name]

Name:	
Description:	
Use Cases:	
Features:	

Quality Attributes

This section describes the quality attributes for each sub-module.

Other Views

This section, if applicable, shows other views for the domain architecture.

Quality Attributes

This section presents the quality attributes for the architecture.

Component Diagrams

This section shows the components, their interfaces and communication among them.

Appendix H. Questionnaires used in the Experimental Study

This appendix presents the two questionnaires used in the experimental study.

QT1 – INDIVIDUAL QUESTIONNAIRE FOR THE PARTICIPANTS OF THE EXPERIMENT

ID: ___

Used technique: () RiSE Process

Domain: **Date**:

Course: ( ) Computer Science ( ) Informatics Bachelor ( ) Data processing
( ) Other _____

Degree: ( ) Graduate ( ) M.Sc. ( ) Ph.D ( ) Specialization

Period: () Daytime ( ) Nighttime Current period: _____ in a total of _____ semesters

In which of the categories below do you belong, in relation to domain engineering projects?

( ) I have no experience in projects that involve domain analysis.
( ) I have no experience in projects that involve domain design.
( ) I have no experience in projects that involve domain implementation.

( ) I have developed some projects of this kind during graduation/post-graduation courses, using domain analysis.
( ) I have developed some projects of this kind during graduation/post-graduation courses, using domain design.

( ) I have developed some projects of this kind during graduation/post-graduation courses, using domain implementation.

( ) I have developed, professionally, some projects using the domain analysis approach (up to 3).
( ) I have developed, professionally, some projects using the domain design approach (up to 3)
( ) I have developed, professionally, some projects using the domain implementation approach (up to 3)

( ) I have developed, professionally, several projects using the domain analysis approach (more than 3).
( ) I have developed, professionally, several projects using the domain design approach (more than 3).
( ) I have developed, professionally, several projects using the domain implementation approach (more than 3).

( ) Other, specify: _____

Please, inform which courses you attended in the system analysis / software engineering / software reuse areas

_____
_____
_____
_____
_____

Do you know domain analysis techniques? Which one(s)?

_____

Do you know domain design techniques? Which one(s)?

_____

Do you know domain implementation techniques? Which one(s)

_____

Check you experience or the activities (positions) that you exercise (or have exercised), in the software development area:
( ) Systems analyst
( ) Domain analyst
( ) Software architect
( ) Software engineer
( ) Components developer
( ) Applications developer (with components)
( ) Tests engineer
( ) Quality engineer
( ) Configuration engineer
( ) Project manager

( ) Teacher (university) in informatics (reuse-oriented disciplines)
( ) Others: _____

In how many developments of application using some reuse technique have you participated?

Large complexity:

() None ( ) 1 - 2 ( ) 3 - 7 ( )More than 7

Medium complexity:

() None () 1 - 2 ( ) 3 - 7 ( )More than 7

Small complexity:

() None ( ) 1 - 2 ( ) 3 - 7 ( )More than 7

List the application domains of the projects that you have participated using reuse concepts and techniques (In decreasing order of experience)

(Examples of application domains: educational, hospital, administrative, financial, scientific)

_____

_____

_____

If it is the case, in which phases of the development life cycle of reusable artifacts have you participated?

( ) Requirements specification ( ) Design ( ) Implementation

( ) Testing ( ) Maintenance ( ) Other (specify):_____

How would you classify your understanding in relation to software reuse (its concepts, objectives and usage viability)

( ) Excellent ( ) High ( ) Good ( ) Average ( ) Low ( ) None

Informing the amount of training on reuse you have, by checking the correspondent items and quantities below (excluding the course ministered in this semester)

Courses (up to 8 hs):

() None ( ) 1 - 2 ( ) 3 - 7 ( ) >7

Courses (up to 40 hs):

() None ( ) 1 - 2 ( ) 3 - 7 ( ) >7

Courses (more than 40 hs):

() None ( ) 1 - 2 ( ) 3 - 7 ( ) >7

Symposiums/Conferences:

( ) None () 1 - 2 ( ) 3 - 7 ( ) >7

Publications of national papers:

( ) None () 1 - 2 ( ) 3 - 7 ( ) >7

Publications of international papers:

( ) None () 1 - 2 ( ) 3 - 7 ( ) >7

Others: _____

[1]Which area are you most interested in:

( ) Software Engineering ( ) Networking/Distributed Systems ( ) Databases
( ) Artificial Intelligence ( ) Hypermedia ( ) Computers architecture
( ) Graphical Computing ( ) Other: _____

Observations or comments: (please use the back page if the space below is insufficient)

_____
_____
_____
_____

Time sheet for the project development

Iteration	Date	Start	End	Time (man/hour)	Activity

Activities summary

Iteration	Activity	Total time (hours)

[1] You may also define sub-areas associated to the selected areas. For example: Area (Software Engineering), Sub-area (Software reuse, Model driven development, etc).

QT2 – INDIVIDUAL QUESTIONNAIRE FOR THE PARTICIPANTS OF THE EXPERIMENT

ID: ____

Used technique: () RiSE Process

Regarding the project, answer:

Did you have any difficulties in understanding the project? Which one(s)?

Additional comments

_____
_____
_____
_____

Regarding the analysis, design and implementation using the RiSE process, answer:

Which difficulties did you find in the analysis (justify)?

_____
_____
_____
_____

Which improvements would you suggest for the domain analysis phase?

_____
_____
_____
_____

Which difficulties did you find in the design phase (justify)?

_____
_____
_____
_____

Which improvements would you suggest for the domain design phase?

_____
_____
_____
_____

Which difficulties did you find in the implementation phase (justify)?

_____
_____
_____
_____

Which improvements would you suggest for the domain implementation phase?

_____
_____
_____
_____

What were the biggest difficulties you had to conclude the project?

_____
_____
_____
_____

Other difficulties:

_____
_____
_____
_____