Federal University of Pernambuco
Informatics Center

ADMIRE: Asset Development Metrics-based
Integrated Reuse Environment

Jorge Mascena

A dissertation submitted to the
Faculty of the Informatics Center of the
Federal University of Pernambuco
in partial fulfillment of the requirements
for the degree of Master of Science

Informatics Center

Recife
May, 2006
Contents

Abstract viii
Acknowledgments ix

1 Introduction 1

1.1 Motivation ........................................ 1
1.2 Problem Statement .................................. 4
1.3 Overview of the Proposed Solution ................. 4
1.4 What is Left Out of Scope .......................... 6
1.5 Statement of the Contributions ..................... 7
1.6 Organization of the Dissertation..................... 8

2 Software Reuse 10

2.1 Introduction ...................................... 10
2.2 Types of Reuse .................................... 13
2.3 Maturity Models .................................... 15
2.4 Processes and Frameworks .......................... 16
2.5 Summary ........................................... 19

3 Information Retrieval 22

3.1 Introduction ....................................... 22
3.2 Reuse Repository Systems ......................... 26
3.3 Repository Information Retrieval .................................. 29
3.4 Active Information Retrieval ................................... 31
3.5 Summary ............................................................ 32

4 Reuse Metrics .................................................... 34
  4.1 Introduction .................................................. 34
  4.2 Economics Oriented Metrics ................................. 35
    4.2.1 Software Reuse Impacts ................................ 39
  4.3 Software Structure Oriented Metrics ...................... 40
    4.3.1 Reusability Assessment .................................. 44
  4.4 Reuse Repository Metrics ................................... 44
    4.4.1 Reuse Repository Adoption Strategy ................. 45
  4.5 Summary ...................................................... 46

5 ADMIRE Environment ............................................. 47
  5.1 Requirements ................................................ 47
    5.1.1 Extensibility .............................................. 48
    5.1.2 High Precision .......................................... 49
    5.1.3 Ubiquity .................................................. 49
    5.1.4 Access Control .......................................... 50
    5.1.5 Scalability ............................................... 50
    5.1.6 Dynamicity ............................................... 51
    5.1.7 Continuous Metrics Extraction ....................... 51
    5.1.8 Feedback Support ....................................... 51
    5.1.9 Platform Independence ................................ 52
    5.1.10 Summary of Requirements ......................... 52
  5.2 Technologies .................................................. 52
    5.2.1 Information Retrieval Library ...................... 52
    5.2.2 Content Parsing ......................................... 54
    5.2.3 Integrated Development Environment ............. 55
# List of Figures

1.1 Reusable software location problem (information islands) .................................. 3  
1.2 Structure of the RiSE project ................................................................................. 5  
1.3 The ADMIRE environment architecture .................................................................. 6  

2.1 Software reuse failure modes ................................................................................ 12  

3.1 Information retrieval system architecture ............................................................... 23  
3.2 Contraction and expansion manipulations ............................................................... 25  
3.3 The produce-manage-consume process of reuse repositories .................................. 27  
3.4 Simplified information retrieval process .................................................................. 29  

4.1 Software reuse metrics categories .......................................................................... 36  
4.2 Sample application structure .................................................................................. 41  

5.1 Reuse realization versus reuse potential ................................................................. 57  
5.2 Possible reuse types in a Java program ................................................................... 58  
5.3 Outline of the Reuse Potential extraction process .................................................. 59  
5.4 ADMIRE, the architecture ..................................................................................... 63  
5.5 Outline of initial ADMIRE implementation ......................................................... 68  
5.6 Ant reuse potential extraction task running inside the Eclipse IDE. ...................... 72  
5.7 Eclipse plug-in features screen ............................................................................... 72  
5.8 MSN Messenger robot in action. ............................................................................. 74  

6.1 The known project approach .................................................................................. 77
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.2</td>
<td>Possible pair permutations for projects A and B</td>
<td>80</td>
</tr>
<tr>
<td>6.3</td>
<td>Module level reuse potential matrix for $\text{threshold} = 1.0$</td>
<td>85</td>
</tr>
<tr>
<td>6.4</td>
<td>Module level reuse potential matrix for $\text{threshold} = 0.5$</td>
<td>86</td>
</tr>
<tr>
<td>6.5</td>
<td>Project level reuse potential matrix for $\text{threshold} = 1.0$</td>
<td>86</td>
</tr>
<tr>
<td>6.6</td>
<td>Project level reuse potential matrix for $\text{threshold} = 0.5$</td>
<td>87</td>
</tr>
</tbody>
</table>
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Reuse Dimensions and Corresponding Types</td>
<td>20</td>
</tr>
<tr>
<td>2.2</td>
<td>Main Reuse Maturity Models</td>
<td>21</td>
</tr>
<tr>
<td>2.3</td>
<td>Requirements for effective software reuse processes</td>
<td>21</td>
</tr>
<tr>
<td>4.1</td>
<td>Main economics related software reuse metrics</td>
<td>38</td>
</tr>
<tr>
<td>4.2</td>
<td>Measurable impacts of software reuse</td>
<td>39</td>
</tr>
<tr>
<td>4.3</td>
<td>Amount of reuse metrics</td>
<td>40</td>
</tr>
<tr>
<td>4.4</td>
<td>SampleApp reuse metrics values</td>
<td>44</td>
</tr>
<tr>
<td>5.1</td>
<td>Integrated reuse environment requirements</td>
<td>53</td>
</tr>
<tr>
<td>6.1</td>
<td>Experiment variations</td>
<td>81</td>
</tr>
<tr>
<td>6.2</td>
<td>Statistics of the projects used in the evaluation</td>
<td>83</td>
</tr>
<tr>
<td>6.3</td>
<td>Precision and recall of the information retrieval mechanism</td>
<td>88</td>
</tr>
</tbody>
</table>
Abstract

The challenge for achieving a more efficient software engineering practice comprises a vast number of obstacles related to the intrinsic complexity of software systems and their surrounding contexts. Consequently, software systems tend to fail in meeting the real needs they were developed to address, consuming more resources, thus having a higher cost, and taking longer to complete than anticipated.

The software reuse field is often regarded as the most promising discipline for closing these gaps, however it still fails to fulfill this promise by providing a comprehensive set of models and tools that can be adopted on a systematic fashion. This leads to a poor reuse activity in most software development organizations.

One of the main reasons for the low reuse activity is that developers simply do not attempt to reuse because of a number of reasons, including lack of knowledge about reusable assets and the notion that the cost for reusing is higher than the cost of developing new code.

This work aims at building an integrated reuse environment, the Asset Development Metrics-based Integrated Reuse Environment (ADMIRE), that addresses this problem in two fronts: (1) aiding developers to achieve a higher reuse activity and (2) providing managers with means to monitor the achieved reuse activity and thus take prompt corrective actions.

An active information delivery scheme is used to provide the first part, while a continuous reuse metric extraction mechanism is employed to implement the second part. A new reuse metric is proposed for this purpose and the resulting ADMIRE environment is evaluated with a set of real projects from a software development organizations.
I would like to thank all the people that in some way contributed and helped me accomplishing this important mission in my life. There were so many people that I would not possibly be able to name them all here, so please accept my sincere apologies and be sure that I will always be grateful to all of you.

I certainly would not be able to complete this work without all the emotional support from my family and close friends, especially my parents, for teaching me to always do my best, my loving wife Tacila and my loving son Rodrigo, for being understanding and patient during the whole time of this research.

The RiSE staff, especially Eduardo Almeida and, of course, my visionary adviser Dr. Silvio Meira, have also been of fundamental importance with all the bright ideas and suggestions, guidance, valuable feedback and hard work. I thank them all for the time they took to carefully review this dissertation and attend my preliminary presentations.

I would like to give a special mention to the valuable feedback on the Reuse Metrics Chapter from Dr. Jeffrey S. Poulin, a reuse metrics expert with many well referenced publications including a book entirely devoted to this topic. I was really impressed by how obliging he proved to be.

Last but not least, I would like to thank C.E.S.A.R, the company that I have proudly worked for during the last ten years, for providing me with the time necessary to attend the classes and perform all extra-class activities related to this research. Most importantly, all the context and background knowledge used during this work have been made possible by this excellent company and all the hard working, creative, passionate people that work on it.
Chapter 1

Introduction

A fundamental premise for any type of reuse is the knowledge about the existence of the object of reuse. Such knowledge may already be available, for example, due to the past experience of the subject of the reuse action or may be obtained through knowledge dissemination. Information retrieval is a key mechanism for allowing a uniform dissemination of the knowledge about available reusable objects.

The instantiation of this problem to the software reuse field is the subject of this work. The synergy among the software reuse and information retrieval fields is thoroughly analyzed and exploited in the formulation of an integrated environment that aims at promoting a greater reuse activity level on the quest for developing software with better quality while consuming fewer resources.

This chapter contextualizes the focus of this work and starts by presenting its motivation in Section 1.1 and a clear definition of the problem being tackled in Section 1.2. A brief overview of the proposed solution is presented in Section 1.3, while Section 1.4 describes some related aspects that are not directly addressed by this work. Section 1.5 presents the provided contributions and, finally, Section 1.6 outlines the structure of the remainder of this dissertation.

1.1 Motivation

Software reuse, in all its variances, is generally regarded as the most important mechanism for performing software development more efficiently. This belief has been systematically enforced by empirical studies that have, over the years, successfully demonstrated the effects of reuse on development aspects such as quality

\footnote{Object here is used in a general sense. Not restricted to the object oriented programming paradigm}
CHAPTER 1. INTRODUCTION

and cost of software systems [Lim, 1994] [Henry & Faller, 1995] [Basili et al., 1996] [Devanbu et al., 1996] [Schmietendorf, 1999] [Frakes & Succi, 2001].

Although these studies, in most cases, are restricted to small, domain-specific systems, there is a general notion that such results could somehow be extrapolated to more generic contexts. Unfortunately, as reports from large software development organizations [Griss, 1994] [Joos, 1994] [Griss et al., 1995] and empirical and informal researches [Card & Comer, 1998] [Glass, 1998] indicate, in fact this notion is still far from the truth.

Many reasons have been enumerated [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. The practice, however, has proved this to be a rather complex task, which poses a number of challenges to individual developers, development teams and organizations. Accommodating the different interests of the individuals and groups involved, at different levels, in a typical software development scenario is one of the most challenging tasks [Fichman & Kemerer, 2001].

Yet, most existing solutions on the software reuse field treat reuse as an isolated concept in the development process, therefore failing to provide an effective solution by addressing the reuse problem from an integrated standpoint. From the list of requirements for an effective reuse program [Almeida et al., 2004], two complementary aspects are particularly important and deserve special attention: (1) reusable software location and (2) reuse activity assessment.

Reusable Software Location

The lack of knowledge about available reusable software that is suitable for a particular task combined with the notion of the cost for achieving reuse are the causes for the number one software reuse failure mode [Frakes & Fox, 1995]: no attempt to reuse. As depicted in Figure 1.1, developers face the problem of locating, understanding and incorporated reusable software into newly created software. The available knowledge is usually unstructured and spread in information islands. For this reason, the problem of locating reusable software is a key aspect for an effective software reuse program.

Aggregating reusable software in repositories [Apperly, 2001] is an initial step for facilitating the location problem. However, reuse repositories alone are by no means enough for promoting a greater reuse activity [Frakes & Fox, 1995]. Cultural and cognitive [Ye, 2001] issues still need to be addressed for an effective use of available repositories.

The problem of locating reusable software is then further divided into collecting
1.1. MOTIVATION

The main focus of the reusable software location problem is on the individual developers. The goal is to provide developers with mechanisms that aid in accomplishing development tasks with a higher quality and with less effort.

Reuse Activity Assessment

Once the proper tools for locating reusable software are provided and the individuals are given the conditions for achieving reuse more effectively, the issue becomes assessing the actual achieved reuse activity.

For this purpose, there is a wide range of reuse metrics proposed in the literature [Frakes & Terry, 1996] [Poulin, 2002] [Mascena et al., 2005]. Some metrics operate on the costs involved in the reuse activity while other metrics focus on the amount of reuse from a program structure perspective.

The reuse metrics problem operates on an organizational level, aiming at providing managers with an overall picture of the reuse activity achieved by developers and therefore allowing a more effective management of the development process from the perspective of software reuse.
1.2 Problem Statement

Organizations fail to fulfill the software reuse promise mainly because reuse is viewed as an isolated concept in the development process that can be addressed by merely adopting technological solutions. The consequence is that tasks relative to reuse are often associated with an overhead that is usually considered prohibitive by development teams.

The traditional approach that takes reuse from the perspective of development for and with reuse poses a conflict of interests between the incentives from an organizational standpoint and the incentives from a project team standpoint [Fichman & Kemerer, 2001]. Organizational incentives are related to long term benefits and the overall performance of the organization even at the expense of some ‘unsuccessful’ projects. Project team incentives, on the other hand, are associated with short term benefits and the success of the project itself, like delivering the required functionality within the schedule and the budget.

A proper balance between providing developers with a productive, reuse oriented environment while providing managers the ability to monitor the reuse activity and take corrective actions in the presence of unwanted deviations must be achieved. In this context, the mission of the work described in this dissertation can be stated as:

This work aims at defining and implementing an integrated environment that merges the reuse activities into the normal tasks executed by developers, while providing managers with a unified view of the achieved reuse activity within organizations.

1.3 Overview of the Proposed Solution

In order to accomplish the mission of this work, stated in the previous Section, an Asset Development Integrated Reuse Environment - ADMIRE, is proposed. This Section describes the context of this work and outlines the architecture of the proposed environment.
This work is part of a broader reuse initiative promoted by the Reuse in Software Engineering research group (RiSE) [Almeida et al., 2004]. Figure 1.2 depicts the overall structure of the RiSE project.

The RiSE project addresses reuse aspects not included in the scope of this work, such as reuse processes [Almeida et al., 2005b], component certification [Alvaro et al., 2005] and software reuse environments [Garcia et al., 2006b]. Other tools and environments proposed by the group include, among many others, the Maracatu search engine [Garcia et al., 2006] and the ToolDAy domain analysis tool [Lisboa, 2006].

These efforts are coordinated and will be integrated in a full-fledged enterprise scale reuse solution. The role of the ADMIRE environment on the RiSE project is to provide a metrics-based solution for the assessment of the reuse activity and a framework for the construction of a comprehensive set of tools and mechanisms with the ultimate goal of maximizing the reuse activity.

**Architecture Outline**

The ADMIRE environment consists of a set of core components, integration interfaces and tools that work in conjunction to provide the required functionalities for an effective reuse environment. Figure 1.3 depicts the ADMIRE architecture.
CHAPTER 1. INTRODUCTION

The main components of the architecture are the Searcher, the Extractor and the Indexer. These components provide the core functionality of the environment, while the integration interfaces provide extension points for hooking the ADMIRE environment to existing development and general use tools.

The Searcher component is responsible for finding reuse candidates for a set of artifacts in a repository. The Indexer component is responsible for analyzing the available legacy code and generating an index to be used by the Searcher component. Finally, the Extractor component uses the Searcher component for the calculation of the reuse potential metric. The details of the architecture are described in Section 5.4.

1.4 What is Left Out of Scope

Since the proposed environment is part of a broader context, a set of related aspects will be left out of its scope. Additionally, although the initial set of provided functionalities shall consist on a production quality, ready-to-use environment, they are by no means intended to be exhaustive, solely serving as a solid basis for future enhancements.
1.5. STATEMENT OF THE CONTRIBUTIONS

The aspects not directly addressed by this work are listed below:

- **Existing reuse metrics**: although existing reuse metrics must be thoroughly reviewed during the research phase, the initial version of the proposed environment shall not contemplate their extraction. The rationale is that the available reuse metric extraction tools may easily be incorporated to the environment in future versions;

- **Artifact types**: the main design principle of the environment is extensibility and, although any type of reusable artifacts may be included to the environment, the initial implementation shall only contemplate Java source files; and

- **Functionalities**: some of the functionalities proposed in the architecture, such as the *Crawler* component, shall not be implemented in the initial version of the environment. These functionalities, although important, consist on supporting tools for the core functionalities and thus may be incrementally implemented in future versions of the environment.

1.5 Statement of the Contributions

As a result of the work presented in this dissertation, a list of contributions may be enumerated:

- **Definition of a new reuse metrics category**: based on the research on the reuse metrics field, a gap on the determination of the reuse potential of a set of artifacts in relation to a reuse repository has been identified. A new category of reuse metrics - *reuse potential metrics* - has been defined in order to fill the identified gap;

- **Proposition of a new reuse metric**: besides defining a new reuse metrics category, a new metric on this category - the *reuse potential* - to be used in the ADMIRE environment has been proposed;

- **ADMIRE architecture**: the architecture of an extensible integrated reuse environment has been proposed; and

- **ADMIRE implementation and evaluation**: based on the proposed architecture, an initial implementation of the ADMIRE environment has been provided and evaluated. The provided implementation is flexible enough to be integrated to completely distinct environments, such as the Eclipse IDE and the MSN Messenger.
Besides the final contributions listed above, some intermediate results of this work have been reported in the literature, as shown below:

- A Comparative Study on Software Reuse Metrics and Economic Models from a Traceability Perspective, IEEE Information Reuse and Integration, Las Vegas, USA, 2005;
- The Domain Analysis Concept Revisited: A Practical Approach, International Conference on Software Reuse, Turin, Italy, to appear, 2006;
- Towards an Effective Integrated Reuse Environment, ACM Generative Programming and Component Engineering, Portland, USA, submitted, 2006; and

1.6 Organization of the Dissertation

The remainder of this dissertation is organized as follows:

- **Chapter 2** contains a detailed historical revision of the software reuse field with the goal of identifying the main problems and some solutions adopted in previous works;
- **Chapter 3** reviews the information retrieval field with an emphasis to the instantiation of the information retrieval techniques in the software reuse field;
- **Chapter 4** presents an extensive list of existing reuse metric categories and their corresponding metrics with the goal of providing the basis for the proposed metric to be used in the ADMIRE environment;
- **Chapter 5** describes the ADMIRE environment in detail. The requirements, used technologies, architecture and initial implementation of the environment are discussed;
- **Chapter 6** reports the results of evaluations performed in order to validate the proposed solution and presents the analysis of these results;
• Chapter 7 concludes this dissertation by summarizing the findings of the work and comparing with some related works. Future enhancements to the environment are also discussed and some concluding remarks are presented;

• Appendix A details the ADMIRE internal representation for the reusable assets; and

• Appendix B presents the list of words that are discarded from the reusable assets’ contents by the ADMIRE environment.
Chapter 2

Software Reuse

Since the main motivation of this work is to improve the reuse activity, a proper revision of the concepts involved in the software reuse field is necessary. This chapter contains a detailed description of the software reuse field history and discusses the reasons why, in spite of its promises, it is still not a widespread practice.

The discussion is organized as follows: Section 2.1 presents an introduction to the software reuse field from its inception to the current time, covering the reasons why it is still not widely adopted. The main reuse classifications, according to the approach and the type of software artifacts being reused, are enumerated in Section 2.2. Section 2.3 presents some models for the assessment of the maturity of the reuse practice and Section 2.4 discusses the main reuse processes and frameworks. Finally, Section 2.5 summarizes the main concepts of the software reuse.

2.1 Introduction

The roots of the so called software crisis that emerged at the end of the 1960s are in the difficulty of creating and maintaining complex, large-scale systems in very dynamic contexts while keeping costs low and schedules on time. The fast paced increment of the processing power of available hardware at that time turned the development of very complex software systems possible. Additionally, the cost of hardware was dropping considerably every new generation. These combined factors caused an enormous pressure both on the complexity of software systems that could be developed and on the percentage dedicated to software development on systems’ overall cost [Dijkstra, 1972], resulting in myriad of unsuccessful software projects.

This issue was first addressed at the first NATO Software Engineering Conference in 1968. In his invited paper “Mass Produced Software Components”
2.1. INTRODUCTION

[McIlroy, 1969], McIlroy approached the main concepts of the software reuse field, stating that a sub-industry of reusable software components was necessary to make the software industry well founded and able to tackle the running crisis.

There are different definitions and theories about the origin of the term software engineering as an engineering area, but regardless of these discussions its ultimate goal is to make software development, operation and maintenance more efficient and cost-effective by following similar approaches of more traditional engineering areas [Ghezzi et al., 2002].

Software engineering may be defined as: “(1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. (2) The study of approaches as in (1).”1 From all existing software engineering fields, software reuse is at the same time one of the most promising, controversial and challenging ones. The principle that governs the software reuse discipline in the quest to achieve the general software engineering objective is the idea of leveraging previous knowledge and development efforts into newly developed systems. The Software Engineering Body of Knowledge [SWEBOK, 2004] states that “software reuse is a key factor in maintaining and improving productivity and competitiveness”.

Since the inception of reuse as a software engineering field to the current time, a great amount of effort has been put in both industry and academy. Many case studies, research reports and surveys have published but the truth is that software reuse is still considered to be in its infancy given the absence of industrial large-scale success cases in the literature, as stated by Almeida et al. in [Almeida et al., 2005].

While it is of fundamental importance to analyze the possible reasons for the failure of organizations in fulfilling the reuse promise (among these the lack of well-defined processes, educational programs and tool support) in order to learn from the mistakes and establish improved mechanisms, an important prerequisite is the ability to assess the impacts of reuse-related activities on organization’s production. The ability to measure the impacts of reuse in other software development aspects, especially quality and cost, is a key factor in assessing the level of success achieved by any reuse initiative [Lim, 1994] [Henry & Faller, 1995] [Basili et al., 1996] [Frakes & Succi, 2001] [Mascena et al., 2005].

The advent of new technologies and paradigms, especially the object-oriented paradigm [Budd, 2001], along the evolution of the software reuse field has opened room for an increasing potential of its benefits during the whole life-cycle of a software system and different types of software reuse have been defined in the literature [Frakes & Terry, 1996] to accommodate the possibilities. Section 2.1.1 details the

---

reported reuse types and discusses about possible extensions to the existing type set.

Despite all the benefits reuse may bring, most organizations still struggle to achieve success in their reuse initiatives and many technical, cognitive and social factors have been considered as inhibitors to the reuse practice [Ye & Fischer, 2002]. Although some success cases have been reported in the literature they still represent a small percentage of the software development universe and are mainly related to very limited contexts [Almeida et al., 2005].

From the programmers’ perspective, the main cause for software reuse failure is that developers simply ignore the possibility of trying to reuse existing assets instead of developing new ones from scratch (no attempt to reuse) [Frakes & Fox, 1995], as shown in Figure 2.1. The NIH (Not Invented Here) syndrome is often regarded as the cause for this inhibiting factor, but empirical studies have shown that developers are often willing to reuse if they know the right components to be reused [Lange & Moher, 1989] [Isolda, 1995]. The real problem might be associated with the perception of the cost and the risks associated with reuse, including the location, comprehension, and adaptation of the components to be reused [Ye & Fischer, 2002].

![Software reuse failure modes](image)

Figure 2.1: Software reuse failure modes [Frakes & Fox, 1995]

The reasons for the failure of systematic reuse in organizations are discussed in [Fichman & Kemerer, 2001]. Fichman and Kemerer present a study that focuses on the general perception that systematic reuse programs with long-term organization-wide objectives are intrinsically incompatible with project team priorities, such as accomplishing project completion on time and within budget.
2.2 TYPES OF REUSE

According to the Fichman and Kemerer, this inaccurate perception is the main reuse inhibitor on organizations with dispersed and autonomous software teams. The risks and costs involved with unsuccessful reuse initiatives as another important reuse inhibitor, a frequently neglected subject, are also discussed. He proposes a set of options for a reuse program that includes the structure of the organization, a process model and a funding model. The main goal is to maximize the probability of benefits for individual project teams by also considering less sophisticated forms of reuse instead of just seeking for an inter-organizational “as-is” reuse. This would help reducing the costs and risks while preserving compatibility with short-term incentives of project teams.

2.2 Types of Reuse

Software reuse may be accomplished in different forms. Some forms are easier to be carried out and require less overhead on the development process but have smaller impacts while other forms require additional effort and a disciplined approach but usually have a higher potential payoff. Table 2.1 summarizes the different reuse dimensions and corresponding types [Frakes & Terry, 1996].

Most reuse programs rely on a combination of techniques from different dimensions. The most frequently reported combination is systematic, black box reuse. As pointed out in [Fichman & Kemerer, 2001], this combination has associated risks and costs that tend to inhibit the reuse practice in organizations.

There is some controversy about the different reuse classifications. Some authors consider that reuse should only be accounted when the reused parts are external to the product being developed [Poulin, 1997], thus making the development scope dimension invalid. The same applies to the modification dimension: sometimes, just black box reuse is considered as valid reuse [Poulin, 1997].

It is important to stress that the reuse types listed in Table 2.1 also apply to different assets other than source code [Leach, 1997] [Lim, 1998]. Actually, early life cycle stages assets reuse have potentially greater impacts on the outcome of the development process [Neighbors, 1980]. The rationale is that the sooner a reuse opportunity is realized, the less duplicated effort will be wasted during the whole product life cycle. If a new application’s functional requirement, for example, has already been implemented by an existing application, chances are the same solution will apply to the new situation and all generated assets will potentially be reused for the new application. Different reuse processes try to explore this by defining reuse checkpoints from the start of a product development [Almeida et al., 2005b].

An important classification of the approaches for achieving software reuse is
presented in [Sommerville, 2004]. Some of these approaches have commonalities with some of the reuse types presented in [Frakes & Terry, 1996]. These approaches include:

- **Design Patterns**: generic abstractions, reusable in different contexts, that show interactions between objects;

- **Component-Based Development (CBD)**: system development is done by integrating reusable collections of objects (components);

- **Application Frameworks**: reusable collections of classes that may be adapted and extended to create applications;

- **Legacy System Wrapping**: definition of a set of interfaces to reusable legacy systems;

- **Service-Oriented Systems**: reusable services attached to application systems at runtime;

- **Application Product Lines**: reusable base architecture for similar products that may be adapted to different contexts;

- **COTS Integration**: reusable commercial off-the-shelf components, usually provided by external organizations, are integrated to applications;

- **Configurable Vertical Application**: configurable generic systems that may be reused in different contexts;

- **Program Libraries**: reusable implementations of common operations available as classes or functions;

- **Program Generators**: systems that reuse patterns and algorithms, usually through the use of templates, to generate applications for specific domains; and

- **Aspect-Oriented Software Development**: reusable crosscutting concerns that are inserted into applications’ code during compile time.

From all reuse classifications available in the literature, one can see that organizations may reach different software reuse maturity levels depending on the extent of reuse achieved, the types of assets being reused and the approaches adopted, among other factors. The boundaries of the different levels of maturity are established by reuse maturity models.
2.3 Maturity Models

Reuse maturity models basically aim at assessing how systematic reuse activities are in an organization. Depending on the model, different factors are considered in determining an organization’s reuse maturity level and a variable number of levels exist.

From the vast number of reuse maturity models proposed over the years [Koltun & Hudson, 1991] [Cusumano, 1991] [Davis, 1992] [SPC, 1992] [Davis, 1993] [Sindre et al., 1995] [Frakes & Terry, 1996] [Bassett, 1997] [Lim, 1998] [Putnam & Myers, 2003], some are inspired by the Capability Maturity Model (CMM) at the Software Engineering Institute (SEI) at Carnegie Mellon University [CMM, 1995] and define five levels of maturity. Other models diverge from CMM by defining between four and six stages reflecting the progression of reuse activities.

Koltun and Hudson [Koltun & Hudson, 1991] were the first to propose a reuse maturity model inspired by the success achieved by SEI’s CMM in establishing an association in people’s minds between the quality of products and the quality of the processes used to develop those products and, more importantly, convincing executive managers that process improvement must be conducted gradually, building on strong process improvement foundations. Their ultimate goal was to extend these notions by incorporating the concepts of software reuse.

The model defines five levels: Initial/Chaotic, Monitored, Coordinated, Planned and Ingrained. Along with the levels, ten maturity dimensions were defined and a correspondence between the levels and dimensions was established. For each one of the ten dimensions, an organizational would fit in one of the five levels. The authors identified several cultural, institutional, financial, technical and legal obstacles for the systematic reuse adoption in organizations during the development of the model. Although no formal study had been conducted at the time, the authors estimated that most organizations would fall into the Initial/Chaotic level.

After this initiative, a spur of research in this topic produced many other reuse maturity models [Cusumano, 1991] [Davis, 1992] [SPC, 1992] [Davis, 1993] [Sindre et al., 1995] [Bassett, 1997] [Putnam & Myers, 2003] that had many overlapping points. Table 2.2 summarizes the main reuse maturity models.

The main problem with all the reuse maturity models proposed so far is that there is no formal study assessing the maturity level of organizations using such models. Furthermore, researchers must agree on a common reuse maturity model so it can be effectively used as a benchmark among organizations. One possible solution may be achieved by incorporating reuse specific aspects into more general maturity models such as CMM and CMMI.
Regardless of the maturity model adopted, organizations must set up a development environment that favors reuse in a systematic manner. Reuse processes and frameworks are at the core of such environments and are described next.

2.4 Processes and Frameworks

Since processes capture the best practices for specific problems and allow a fast dissemination of such practices, they are natural candidate instruments for a successful systematic reuse adoption program. In the search for improved reuse practices that maximize the positive impacts of reuse in organizations, different reuse processes and frameworks have been proposed in the literature.

A comprehensive survey on the existing reuse processes is presented in [Almeida et al., 2004]. Almeida et al. state that most reuse processes proposed so far present fundamental gaps in crucial activities, such as development for and with reuse, and define a set of requirements for an effective reuse process that are summarized in Table 2.3. The definition of a set of metrics is of fundamental relevance to the success of a reuse process or framework.

Although alternative approaches, such as Component-Based Development (CBD) or Component-Oriented Software Engineering (COSE) [Dogru & Tanik, 2003], have been proposed as solutions for improved reuse activity in organizations, current software reuse processes are divided into two main categories: Domain Engineering and Software Product Lines.

Domain Engineering

Domain Engineering consists on collecting, organizing and storing past experience in a particular domain in the form of reusable assets along with mechanisms for reusing those assets [Czarnecki & Eisenecker, 2000].

According to the first contribution to the domain engineering field, proposed in 1980 by Neighbors [Neighbors, 1980], efficient low-level executable programs could be derived from high-level, domain-specific languages. The derivation process included four distinct phases: (1) the analysis of a domain, (2) the formulation of a model of a domain into a domain language, (3) the use of software components to implement domain languages and (4) the use of program transformations to specialize components for specific systems.

Domain analysis is concerned with all possible actions in a particular problem area. In this sense, domain analysis is much more complex than system analysis.
since it may involve the development of a general model of the objects in the domain. It requires a greater extent of knowledge on the domain under analysis and a considerable effort in order to cover all, or at least the most common, possibilities.

Domain languages are defined as high-level specific languages with the purpose of being easy to read for specialists in the domain at hand. Objects in the domain language represent objects in the problem area (domain) and operations in the domain language represent operations in the problem area.

Software components describe the semantics of objects or operations in a specific domain. Components must be kept small in size in order to be flexible enough to permit different specializations for applications in the specific domain.

Source-to-source program transformations remove unnecessary generalities from software components for specific applications and bind these components together with the goal of making efficient component-based systems.

Despite the importance of this work for the domain engineering field, the complexity of the proposed approach, including activities such as writing transformations, makes it difficult to be adopted in the industrial environment. Some extensions to this work have been proposed [Leite, 1994] [Sant’Ana et al., 1998] but the main issues regarding the complexity of the approach remain open.

A number of related works in the domain engineering field have been proposed after the initial contribution from Neighbors. In 1993, the Software Technology for Adaptable, Reliable Systems - STARS - developed a set of concepts, processes, methods and tools.

The Conceptual Framework for Reuse Processes - CFRP - served as a conceptual foundation for domain-specific reuse [STARS, 1993a], being too generic to be directly adopted. The Reuse-Oriented Software Evolution - ROSE - process model is based on CFRP [STARS, 1993b] and bridges the gap between a conceptual framework and a process model by partitioning software development into domain engineering, asset management and application engineering and defining generic activities for integrating software maintenance and reengineering in the context of domain-specific reuse. The generality of the activities, making it hard to figure out how to perform them, is the main problem with the ROSE process.

The Organization Domain Modeling - ODM - method is compatible with CFRP and was developed with the goal of achieving a systematic transformation of artifacts from legacy systems into reusable assets in a formal, manageable and repeatable manner. As with the previous approaches, the main problem of this method is the generality of the proposed activities and guidance.

Besides the efforts of the STARS program, concurrent, and sometimes complementary, domain engineering approaches have been proposed [Kang et al., 1990]
CHAPTER 2. SOFTWARE REUSE

[Jacobson et al., 1997] [Griss et al., 1998] [Kang et al., 1998] but in general they all suffer from the lack of a detailed definition of the activities to be performed, as stated in [Almeida et al., 2005b].

Software Product Lines

The concept of Software Product Lines [Clements & Northrop, 2001] emerged in the end of the 1990’s as one of the most promising approaches for an effective reuse activity and efficient software development, since it covers a broader range of techniques compared to the domain engineering approach.

The Product Line Software Engineering - PuLSE - methodology, the first work on the software product line, was published by Bayer et al. in 1999 [Bayer et al., 1999]. It implements a framework that supports the introduction of product line development in organizations: customization of the base process, scoping the product line via an economical analysis, modeling the concepts and relationships of the domain of the product line, definition of a base architecture for the product line that covers existing and future applications, instantiation of the product line for product development, and evolution and management of the product line process and artifacts based on a feedback process.

Although this approach has been successfully applied in a variety of contexts [Atkinson, 2000], the components that support the different phases of the product line life-cycle are designed to just provide high-level guidance on the product line development, lacking support for activities related to specification, design and implementation [Almeida et al., 2004]. Furthermore, the PuLSE methodology does not provide an evolutionary approach for introducing product line concepts for existing software products [Simon & Eisenbarth, 2002].

Another problem of the PuLSE methodology is that it does not provide a basic customization process for immature environments where no pre-existing processes exists. For such cases, the cost of introducing the software product line process may be prohibitive. In this context, the KobrA method [Atkinson, 2000] represents a basic customization of the PuLSE methodology. The KobrA method addresses reuse-in-the-small by employing component-based development methods and targets reuse-in-the-large by making use of software product line methodologies. The main shortcoming of the KobrA method is that it lacks guidance on specific activities such as domain analysis and domain design.

Different software product lines related approaches have been proposed during the last years [America et al., 2000] [Kang et al., 2002] [Northrop, 2004], but none of them has a detailed definition of important activities such as domain analysis and
product development. Different research groups\footnote{Including the RiSE research group (\url{http://www.rise.com.br})} have studied this subject and new approaches will soon be available [Almeida, 2007].

2.5 Summary

The software reuse field has been investigated for decades, but still, existing reuse processes are distant from the current development practice in organizations. This poses a great overhead and inhibit the adoption of such processes. Consequently, few success cases in very specific contexts have been reported in the literature.

As briefly described in this Chapter, this vast research area involves aspects such as \textit{methods, metrics, processes, frameworks, environments} and \textit{tools}. These and other aspects related to the state-of-the-art in the software reuse field are thoroughly discussed in [RiSE, 2006].

A prerequisite for reuse is the knowledge about the reusable assets. This knowledge can be obtained through experience or disseminated and such dissemination depends on an efficient access to the available information about previous experiences. The next Chapter discusses a field intrinsically related to software reuse that was also very important for the definition of the proposed solution: \textit{information retrieval}.
Table 2.1: Reuse Dimensions and Corresponding Types [Frakes & Terry, 1996]

<table>
<thead>
<tr>
<th>Development Scope</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internal</td>
<td>Reuse of parts within the same product [Fenton, 1991].</td>
</tr>
<tr>
<td>External</td>
<td>Reuse of externally constructed parts in a product [Fenton, 1991].</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Modification</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>White Box</td>
<td>Reuse with modification [Prieto-Diaz, 1993].</td>
</tr>
<tr>
<td>Black Box</td>
<td>Reuse without modification [Prieto-Diaz, 1993].</td>
</tr>
<tr>
<td>Glass Box</td>
<td>Reuse by example [Prieto-Diaz, 1993].</td>
</tr>
<tr>
<td>Adaptive</td>
<td>Uses large structures as invariants and restricts variability to isolated parts [Barnes &amp; Bollinger, 1991].</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Approach</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generative</td>
<td>Reuse at the specification level with application or code generators [Prieto-Diaz, 1993].</td>
</tr>
<tr>
<td>Compositional</td>
<td>Use of existing components as building blocks for new systems [Prieto-Diaz, 1993].</td>
</tr>
<tr>
<td>In-the-Small</td>
<td>Use of components which dependent on the environment of the application for full functionality [Favaro, 1991].</td>
</tr>
<tr>
<td>In-the-Large</td>
<td>The use of large, self-contained packages such as spreadsheets and operating systems [Favaro, 1991].</td>
</tr>
<tr>
<td>Indirect</td>
<td>Reuse through an intermediate entity or a set of intermediary entities [Bieman &amp; Karunanithi, 1993].</td>
</tr>
<tr>
<td>Direct</td>
<td>Reuse without going through an intermediate entity [Bieman &amp; Karunanithi, 1993].</td>
</tr>
<tr>
<td>Carried Over</td>
<td>A version of a software component taken to be used in subsequent versions of the same system [Ogush, 1992].</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Domain Scope</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertical</td>
<td>Reuse within a domain [Prieto-Diaz, 1993].</td>
</tr>
<tr>
<td>Horizontal</td>
<td>Reuse of generic parts [Prieto-Diaz, 1993].</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Management</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Systematic</td>
<td>Formal practice of reuse [Prieto-Diaz, 1993].</td>
</tr>
<tr>
<td>Ad hoc</td>
<td>The selection of components which are not designed for reuse from general libraries [Prieto-Diaz, 1993].</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Reused Entity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract Level</td>
<td>High-level abstractions within an object-oriented inheritance structure [McGregor &amp; Sykes, 1992].</td>
</tr>
<tr>
<td>Instance Level</td>
<td>Defined as simply creating an instance of an existing class [McGregor &amp; Sykes, 1992].</td>
</tr>
<tr>
<td>Customization</td>
<td>Use of object-oriented inheritance to support incremental development [McGregor &amp; Sykes, 1992].</td>
</tr>
<tr>
<td>Reuse</td>
<td>Reuse of generic packages, such as package templates or subprograms [Bieman &amp; Karunanithi, 1993].</td>
</tr>
<tr>
<td>Generic</td>
<td>Reuse of generic packages, such as package templates or subprograms [Bieman &amp; Karunanithi, 1993].</td>
</tr>
<tr>
<td>Source Code</td>
<td>Low-level modification of an existing object-oriented class to change its performance characteristics [Frakes &amp; Terry, 1996].</td>
</tr>
</tbody>
</table>
### Table 2.2: Main Reuse Maturity Models

<table>
<thead>
<tr>
<th>Level</th>
<th>Koltun &amp; Hudson</th>
<th>Cusumano</th>
<th>SPC</th>
<th>REBOOT</th>
<th>Putman &amp; Myers</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Initial / Chaotic</td>
<td>None</td>
<td>Ad-hoc</td>
<td>Chaotic</td>
<td>No Reuse</td>
</tr>
<tr>
<td>2</td>
<td>Monitored</td>
<td>Some</td>
<td>Repeatable</td>
<td>Repeatable</td>
<td>Hip-Pocket Reuse</td>
</tr>
<tr>
<td>3</td>
<td>Coordinated</td>
<td>More</td>
<td>Portable</td>
<td>Defined</td>
<td>Repository Reuse</td>
</tr>
<tr>
<td>4</td>
<td>Planned</td>
<td>Most</td>
<td>Architectural</td>
<td>Managed</td>
<td>Product-Line Reuse</td>
</tr>
<tr>
<td>5</td>
<td>Ingrained</td>
<td>Systematic</td>
<td>Optimized</td>
<td>Enterprise Resource Planning</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Architecture-Wide Reuse</td>
</tr>
</tbody>
</table>

### Table 2.3: Requirements for effective software reuse processes [Almeida et al., 2004]

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Development for reuse</td>
<td>Methods to enforce the production of assets which are reusable in future products.</td>
</tr>
<tr>
<td>Development with reuse</td>
<td>Methods to identify requirements, locate reusable assets, make adaptations and integrate these assets into new products.</td>
</tr>
<tr>
<td>Metrics</td>
<td>Definition of what, where and when to measure. Fundamental for assessing the effectiveness of the reuse initiative.</td>
</tr>
<tr>
<td>Costs</td>
<td>Methods to estimate the additional cost incurred by development for reuse and savings from development with reuse.</td>
</tr>
<tr>
<td>Reengineering</td>
<td>Reuse of knowledge embedded in legacy systems in the form of reconstructed code and documentation.</td>
</tr>
<tr>
<td>Adaptation</td>
<td>The ability to adapt the process according to the context where it will be adopted.</td>
</tr>
<tr>
<td>Software reuse environment</td>
<td>Comprehensive set of tools that support activities defined in the process.</td>
</tr>
</tbody>
</table>
Chapter 3

Information Retrieval

Software reuse is largely influenced by the availability and knowledge of assets constructed for previous systems during the development of new systems. In this context, information retrieval mechanisms play a very important role by providing access to the available assets in a structured manner.

This chapter presents a review on the information retrieval field with emphasis to the application in the software reuse context and is organized as follows: Section 3.1 brings an introduction to the information retrieval field. Reuse repository systems are introduced in Section 3.2 while the intersection between reuse repositories and information retrieval is discussed in Section 3.3. Information retrieval schemes enhanced with active information delivery are presented in Section 3.4 and Section 3.5 summarizes the discussion on information retrieval and its application to the software reuse context.

3.1 Introduction

Access to information has always been of fundamental importance to human activities and the time needed for processing information is proportional do the volume of available data. Since large sets of data are unsuitable for direct manipulation in reasonable time frames, specialized data structures are created for representing such data, thus allowing a faster access to the contained information. The most common data structure used for this purpose is known as index [Baeza-Yates & Ribeiro-Neto, 1999].

The preparation of the information for later access usually follows an information retrieval process. A general purpose retrieval process includes pre-retrieval tasks, such as the selection, generation of a logical view and indexing of the text of
3.1. INTRODUCTION

the documents to be used in the retrieving process, and retrieval tasks, such as the generation of a representation of the user query, processing of the query and ranking of the results according to its likelihood or relevance to the query. A simplified software architecture to achieve this process is shown in Figure 3.1.

Figure 3.1: Simplified architecture of an information retrieval system [Baeza-Yates & Ribeiro-Neto, 1999]

Information retrieval has an intrinsic difficulty associated with the subjective concept of the likelihood or relevance of a document in the context of a given query, as opposed to data retrieval, which is concerned with retrieving all data that matches a given query, generally expressed in a regular expression or in a relational algebra expression [Baeza-Yates & Ribeiro-Neto, 1999]. A detailed comparison between information retrieval and data retrieval is presented by Rijsbergen in [Rijsbergen, 1979].

According to Ye [Ye, 2001], the main models to index and retrieve free-text information are vector space model (VSM) and probabilistic model.

The vector space model represents queries and documents as vectors of terms of the entire document collection. For each document, the value of the terms in the corresponding vector represent the importance of that term to the document. Variations of the model represent the values of the vectors in different ways. Common variations are binary value model, term frequency and inverse document frequency [Ye, 2001].
The probabilistic model makes use of formal theories of probability and statistics to estimate the relevance of documents to a given query [Ye, 2001].

The results retrieved by queries may be influenced by a number of manipulations to the data set and the queries themselves. Expansion manipulations operate on the raw data in order to widen the range of retrievable query results. Examples of such manipulations are insertion of word synonyms to capture similar concepts, word stemming to capture variations of the same word, and word phonetic representation to capture incorrectly spelled words.

Contraction manipulations, on the other hand, operate on the raw data in order to narrow the range of retrievable query results. Examples of such manipulations are document type matching to restrict results to certain document types and signature matching to filter out results that do not match certain signature patterns.

Information retrieval mechanisms must balance expansion and contraction manipulations according to their specific goals. Expansion manipulations are usually employed when the average number of search results is small, while contraction manipulations are used when the average size of the result sets is large. Figure 3.2 illustrates the manipulation types and their effects to the query result set.

In this example, the balancing between contraction and expansion manipulations is stressed: if the goal is to retrieved all related aspects of the submitted query (A.java) with the java type (including synonyms), the alternative that yields the best precision is combining the synonym expansion with the type contraction manipulation. With this combination, only A.java and B.java are retrieved, which is exactly the intended result.

The ranking of the results of a query in an information retrieval system is performed by a ranking function, usually based on the relevance or likelihood of the documents in relation to the executed query. The ranking function may also take into account, besides the similarity between the query and the documents, the importance of the document in the repository. PageRank is an example of a ranking function that is based on the number of references to the documents in a repository [Page et al., 1998] used in web search engines such as Google[^1] [Brin & Page, 1998]. In this case, the documents are web pages and the references are hyperlinks between the pages.

Two fundamental metrics are conventionally used to assess the efficiency of any information retrieval mechanism: recall and precision [Cleverdon et al., 1966]. The capacity of a mechanism of retrieving relevant results is measured by the recall metric \( r \), while the capacity of avoiding irrelevant results is measured by the precision metric \( p \). The following equations are used to calculate these metrics:

[^1]: http://www.google.com
3.1. INTRODUCTION

Figure 3.2: Example of contraction and expansion manipulations. $A$ and $B$ are synonyms.

\[(3.1)\]
\[
r = \frac{a}{a + b}
\]

\[(3.2)\]
\[
p = \frac{a}{a + c}
\]

Where $a$ is the count of relevant documents retrieved, $b$ is the count of relevant documents missed, and $c$ is the count of irrelevant documents retrieved.

Any retrieval mechanism must balance the results of these two metrics based on its specific goals. A high precision mechanism will be stricter when matching results, usually discarding potentially relevant results. On the other hand, a high recall mechanism will relax the matching rules, usually retrieving results that are not relevant to the context of a given query. Expansion and contraction manipulations are usually employed for this end.

The harmonic mean of these two metrics, called $F_1$-Measure [Rijsbergen, 1979], is also a frequently used performance assessment metric for information retrieval mechanisms. This metric equally balances recall and precision (thus, the ‘1’ in $F_1$-Measure) and a more generic metric, $F_N$-Measure, may be instantiated with different
‘N’ values to favor recall or precision, depending on the goal of the measurement. According to Yang and Liu [Yang & Liu, 1999], this measure was first introduced by Rijsbergen [Rijsbergen, 1979] and its calculation uses the following equation:

\[ F_1(r, p) = \frac{2rp}{r + p} \]

Where \( r \) is the recall and \( p \) is the precision of an information retrieval mechanism for a given query.

Besides the aforementioned metrics, the performance of the mechanism in retrieving the results for the user queries and the space required by the indexes used by the mechanism are common non-functional aspects considered for the evaluation of information retrieval mechanisms. However, since the benefits of a high-precision/high-recall retrieval mechanism in promoting reuse are evident and the amount of data manipulated by such systems is not comparable to the amount of data manipulated by information retrieval systems such as web search engines [Brin & Page, 1998], the non-functional aspects will be left out of the scope of this research.

### 3.2 Reuse Repository Systems

The task of reusing software has a prerequisite that the appropriate components or artifacts are available somewhere and that they can be somehow located. Reuse repository systems support the location of reusable software artifacts. In fact, most of the effort put on reuse repository systems has been directed to component searching and retrieving [Mili et al., 1998] because of the synergy between the information retrieval and software reuse fields, as stated in Chapter 1.

All existing reuse-based processes discussed in Section 2.2 rely on the central role of reuse repository systems as convergence points (or communication buses) and management assistants of the reuse activity in organizations [Apperly, 2001]. Basically, three distinct activities are observable in such processes: (1) production of reusable components (development for reuse), (2) management of the repository system and quality control and (3) consumption of reusable components (development with reuse). The whole produce-manage-consume cycle is summarized in Figure 3.3. From these 3 parts, the consumption and management parts are of special interest to this work, since information retrieval and reuse potential are more related to the reuse repository and its consumers consumers.

The production tasks are intrinsically related to processes and practices that
3.2. REUSE REPOSITORY SYSTEMS

enforce the reusability of the produced artifacts. Section 2.4 discusses these processes in more detail.

The management tasks are related to the repository itself and the artifacts that populate it. Possible examples of such tasks are artifact quality assurance [Alvaro et al., 2005] and repository metrics (as discussed in Section 4.4) collection.

From the consumption perspective, besides the traditional methodology centered paradigm of development with reuse, where programmers must adjust to the methodology in order to be able to consume reusable assets from the repository, there is a recent research trend based on the concept of user-centered (or in this case, programmer-centered) paradigms [Jarzabek & Huang, 1998] of reuse within development, where the methodology is meld into the normal activities of programmers in an attempt to make it more appealing to them [Winograd & Flores, 1986] [Aaen, 1992] [Ye, 2001] and consequently, more effective. Section 3.2 discusses this issue in more detail.

Although most of the focus on the reuse repository systems is on information retrieval, the use of such systems has additional management related benefits, including the metrics extraction of the reuse activity [Frakes & Terry, 1996], the quality certification of the reusable artifacts [Alvaro et al., 2005] and incentives for

Figure 3.3: The produce-manage-consume process of reuse repositories
the production and consumption of reusable artifacts [Fichman & Kemerer, 2001].

In a study conducted by Banker et al. [Banker et al., 1993], it was observed that repository-based CASE environments are a viable solution for a systematic collection of metrics of the reuse activity at the repository level, assessing the efficiency of the repository. Additionally, the collected metrics may be used to determine the objects that are most likely to be reused and the contexts where reuse is most likely to occur.

These additional benefits help mitigating some risks associated with the reuse activity [Frakes & Fox, 1995] [Rashid & Kotonya, 2001] [Crnkovic & Larsen, 2002]. For example, the certification of the quality of the reusable artifacts in a repository will help reducing the rate of unsuccessful reuse attempts due to artifacts that contain errors or have incomplete documentation.

In this context, reuse repository systems constitute reference points of reusable artifacts with some level of quality control. This is especially important with the increasing widespread availability of open source projects [DiBona et al., 1999] and Commercial Off-The-Shelf (COTS) components [Morisio et al., 2000], to name a few examples of reusable artifacts sources.

Source code is the most common type of artifact to be stored in reuse repository systems, but intermediate products of the software development life cycle, such as models, frameworks, interfaces and patterns, are also considered as important reusable artifacts [Krueger, 1992]. In fact, some advocate that early-stage reuse has potentially more benefits to the development process when compared to source code (implementation) reuse [Neighbors, 1980].

Reusable software components or artifacts are more generally referred to as software assets, which are defined as reusable software units that have a significant value for a company and capture business knowledge [Ezran et al., 2002]. Asset models comprehend the artifacts that are part of the asset contents (source code, design models etc.) and the information provided in the asset meta-data (description, classification etc.).

Although reuse repository systems must fulfill a set of requirements in order to accomplish their purpose [Lucredo et al., 2004], most companies opt for adapting traditional software engineering tools to serve as reuse repositories [Ezran et al., 2002]. Such tools present a set of limitations that minimize the positive impacts of reuse in a software development process.
3.3 Repository Information Retrieval

Information retrieval is traditionally the most explored research topic of reuse repository systems [Mili et al., 1998]. It has a key role in the development with reuse part, or more recently the reuse-within-development part [Ye, 2001], of any reuse-based process.

Instantiating the concepts presented in Section 3.1 to the reuse repository systems context, a document might be any asset or component stored in the repository and the relevance of a component for a given query might represent whether that component may be reused in the situation where the query was performed.

As in any information retrieval process, a typical component retrieval mechanism is composed of a set of steps performed by the reuser [Mili et al., 1995]: (1) understanding and interpretation of the problem, (2) query formulation, and (3) component matching. A simplified version of this process is depicted in Figure 3.4, provided in [Lucredio et al., 2004]. Each of these steps has an associated intrinsic loss of information, which contributes to minimize the overall performance of the retrieving mechanism. For this reason, most of the research effort on the area is focused on minimizing this information loss on each of these steps.

![Figure 3.4: Simplified information retrieval process](https://example.com/figure3.4.png)

Among the works on the problem of repository components classification (in-
two important approaches may be identified: (1) manual and (2) automatic index generation. Manual index generation is an expensive and subjective process, what makes it inappropriate for large collections of dynamic, evolving components [Maarek et al., 1991]. For this reason, automatic index generation techniques are more commonly explored.

Some works propose easing the query formulation stage by representing reusable components as frames and connections between the frames [Henninger, 1997], hierarchical categories [Devanbu et al., 1991], multiple facets [Prieto-Diaz, 1991] or a combination of frames and multiple facets [Ostertag et al., 1992] during the indexing phase. Queries capabilities may be complemented by the classifications and relationships generated by these schemes, allowing users to browse the repository in a structured manner. Since human intervention is required for the categorization, these approaches are usually very expensive to maintain.

For this reason, most reuse repository systems use free-text indexing [Maarek et al., 1991] [Girardi & Ibrahim, 1995] [Etzkorn & Davis, 1997] [DiFelice & Fonzi, 1998] [Michail & Notkin, 1999] because of its simplicity. Moreover, studies have shown that despite their simplicity, free-text indexing repository systems have an equivalent performance of more elaborate, effort-consuming repository systems [Frakes & Pole, 1994] [Mili et al., 1997].

Components constraints, such as signature matching and formal specification, are used by some reuse repository systems for component retrieval [Rittri, 1989] [Stringer-Calvert, 1994] [Zaremski & Wing, 1995] [Ye, 2001]. This approach is generally too restrictive and difficult for programmers to formulate queries, but has been successfully used as an automatic filter for query results from free-text indexing mechanisms [Ye, 2001].

An even more restrictive component retrieval approach, based on both the signature and the behavior of the component, has been proposed [Hall, 1993] [Podgurski & Pierce, 1993]. The user specifies a signature and a set of inputs and expected outputs. The components with matching signatures are executed with the given inputs and the outputs are compared. The shortcomings of this approach are twofold: formulating queries is expansive and the matching mechanism is way too restrictive (i.e. a very powerful contraction manipulation), potentially missing close enough components in many situations [Ye, 2001].

Some schemes have also been proposed out of the traditional browsing/querying box of most information repository systems in order to improve their overall information retrieval efficiency. Query by reformulation [Williams et al., 1982] [Fischer & Nieper-Lemke, 1989] [Henninger, 1993], information filtering [Belkin & Croft, 1992], Latent Semantic Analysis (LSA) [Landauer & Dumais, 1997] and active information systems [Ye, 2001] are some examples of such schemes.
Query reformulation is the ability information retrieval systems provide users to refine their queries based on the results of previous queries. This concept is based on the fact that most users will have problems in formulating a well-defined query in their first attempt because the lack familiarity with the structure and representation of the system [Jones, 1997].

Information filtering is employed in order to improve the relevance of information by defining a set of rules for discarding irrelevant information. This is especially important in situations when it is easier to identify what is not relevant than to identify what is actually relevant.

Latent Semantic Analysis is an extension of the vector space model [Salton, 1975], which represents documents and queries as vectors of terms contained in the whole document collection and similarity as a function of the word overlapping of these vectors. The most common similarity measure is the Cosine coefficient of the angle between the vectors, but other measures, such as Inner Product, Jaccard and Dice coefficients [Salton, 1988], have also been proposed.

Active Information Systems constitute an important recent trend in the information retrieval field and are discussed in detail in the next section.

## 3.4 Active Information Retrieval

Traditional information retrieval systems or, in the software reuse context, active reuse repository systems, may be seen as passive engines that provide information upon explicit user requests (queries). Thus, the user is responsible for having the initiative of searching or browsing the system in an articulated way in order to retrieve and understand, and possibly reusing, the desired information [Montgomery & Ruspini, 1981].

The problem with this scenario is that users usually either ignore the information that is available in the system or do not know how to articulate queries in order to retrieve the desired information [Ye, 2001]. In the reuse repository systems context this adds up to the cost of reuse, contributing to the number one reuse inhibitor: no attempt to reuse [Frakes & Fox, 1995]. Active reuse repository systems tackle this problem by actively delivering reusable artifact candidates given the user context. The challenges in active content delivery include determining what information is relevant to the user context, and when and how to deliver the information [Ye, 2001].

The inference of the information that is relevant to the user in a given context is based on the understanding of the task being performed by the user. This is
accomplished through the interpretation of a shared context between the user and the information system [Winograd & Flores, 1986].

Information may be delivered during the period of time called action-present, while the user is performing a task, or after the task completion. The former approach, known as feedforward [Simon, 1996], has the advantage of actually assisting users to accomplish their tasks, but have to deal with preliminary, imprecise information. The later approach, known as feedback, has the advantage of being potentially more accurate, since there is more information available to be analyzed upon task completion, but may incur in waste of effort to users since possible alternative solutions to the accomplished task will only be available after all the effort has already been put to complete the task. Depending on the duration of the task, the difference between both approaches will be unnoticeable.

The way the information is delivered is also of extreme relevance in active information retrieval systems. Interruptive systems have the advantage of catching user’s attention while his focus is on the task, but may break the work flow of the user. On the other hand, a non-interruptive approach requires the user’s initiative to check the delivered information, which may happen when his or her focus is on a different task, forcing a context switch. The importance of the information to be delivered must be taken into account in order to determine when and how it should be delivered [Sumner, 1995].

Active information retrieval, or Just-in-Time Information Retrieval (JITIR) [Rhodes, 2000], constitutes an important mechanism for augmenting the user experience by providing assistance during task execution. General purpose [Rhodes & Starner, 1996] and web assistants [Lieberman, 1997] [Armstrong et al., 1995] [Balabanovic & Shoham, 1995] are some examples of systems that make use of the active information delivery mechanism.

In the software reuse context, it can be used to increase reuse activity level, especially in environments where no reuse process is defined. For this reason, reuse programs must contemplate information delivery for an incremental reuse approach. Some works have explored information delivery and reuse repositories [Fischer, 1987] [Ye, 2001] for aiding the developer by displaying components that match the current task from a repository. These works focus solely on the developers’ perspective, not constituting on integrated reuse environments.

3.5 Summary

This chapter introduced the information retrieval field and discussed its relationship to software reuse. Reuse repository systems, the convergence point between these
two fields, was briefly discussed. Active, or just-in-time, information retrieval mechanisms where reviewed as an important alternative for improving the effectiveness of the information retrieval related tasks.

The next chapter discusses another important aspect besides information retrieval for an integrated reuse environment: \textit{reuse metrics}.
Chapter 4

Reuse Metrics

As stated in Chapter 2, software reuse metrics are important tools for the assessment of the effectiveness of reuse initiatives. Existing metrics represent economical and technical aspects related to the reuse activity. This Chapter discusses the different types of existing software reuse metrics in more detail and serves as a base for the definition of the metrics to be used in the proposed integrated reuse environment.

The discussion in this Chapter is organized as follows: Section 4.1 briefly introduces the software reuse metrics field and presents a categorization of the existing metrics. Section 4.2 enumerates the economics oriented metrics and presents some studies assessing the impacts of reuse in different development aspects. Section 4.3 illustrates the software structure oriented reuse metrics through a sample application and Section 4.4 briefly discusses some existing reuse repository metrics. Section 4.5 summarizes the discussion on the existing metrics and presents some pointer to the definition of the metrics to be used in the environment.

4.1 Introduction

Since the first time the concept of software reuse was discussed, many intriguing challenges were posed to the software engineering community: if on 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 in the walls of specific domains.

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

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

Another software reuse metrics category, Reuse Repository Metrics (RRM) [Poulin, 1997], is related to reuse repositories that target the assessment of reuse repository aspects such as availability of the repository, search engine performance, quality of available assets and number of times assets were successfully reused. Such metrics can be used as support tools for determining where organizations should direct their effort regarding maintenance and evolution of reusable assets available in the repository and potential new reusable assets to be developed.

Figure 4.1 shows how the reuse metrics field is divided. The remaining of this Chapter will be devoted to describe the state-of-the-art of software reuse metrics in the three main categories.

4.2 Economics Oriented Metrics

Engineering decisions must take into account the economical impacts related to them and metrics are important instruments for assessing such impacts. EORM are mainly concerned with the economical aspects related to reuse programs in organizations and are the basic instruments for organization-wide return on investment models (ROI).

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

EORM are based on a set of observable data [Poulin & Caruso, 1993] [Poulin, 2002]: Shipped Source Instructions (SSI), Changed Source Instructions (CSI), Reused Source Instructions (RSI), Source Instructions Reused by Others (SIRBO) and data related to software development cost and quality, such as software de-
Figure 4.1: Software reuse metrics categories

- **Economics Oriented Reuse Metrics (EORM)** are related to return on investment (ROI) models and the economical impacts of software reuse based on a set of observable data;

- **Software Structure Oriented Reuse Metrics (SORM)** are concerned on what is being reused and how it is being reused from a strictly technical standpoint;

- **Reuse Repository Metrics (RRM)** assess the overall performance of the convergence point of the reuse activities in an organization.
4.2. ECONOMICS ORIENTED METRICS

development cost \((\text{Cost per LOC})\), software development error rate \((\text{Error rate})\) and software error repair cost \((\text{Cost per error})\).

From this set of observable data, the most relevant one is SIRBO which accounts for the amount of code contributed by a team or organization to a reuse program is being actually reused by other teams or organizations. This is very important information for any reuse incentive program or reuse repository metric as discussed in Section 4.4.

Besides the observable data, two additional abstractions must be presented before defining the main economics oriented reuse metrics [Poulin & Caruso, 1993]: Relative Cost of Reuse \((\text{RCR})\) and Relative Cost of Writing Reusable Software \((\text{RCWR})\).

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

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

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

Once the observable data and the basic abstractions are established, the economical value of reuse may be directly derived into the main economics oriented metrics: Reuse Cost Avoidance \((\text{RCA})\), Organizational or Project-level ROI and Reuse Value Added \((\text{RVA})\). Table 4.1 summarizes the definitions of the main economics related reuse metrics.

Reuse Cost Avoidance \((\text{RCA})\) is the cost saved by writing code with reusable assets compared to an estimation of the cost of writing an equivalent code without reusable assets. This metric takes into account not only the development
### Table 4.1: Main economics related software reuse metrics

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

phase, which is usually the case for other defined metrics in the literature, but also the maintenance or service phase. The rationale is that by reusing assets, less code must be produced, thus reducing overall development effort, combined with the fact that a reusable asset has potentially fewer errors than a newly written asset, since it has been tested in previous products using that same asset. Fewer errors mean less time fixing errors and consequently less overall effort during development and maintenance. Suppose the cost for writing and maintaining an application throughout its entire life cycle is estimated in $40K and by reusing available assets that cost drops to $32K, the RCA is $8K. The calculation of RCA is based on RCR.

**Reuse Value Added (RVA)** measures the effectiveness a team or an organization has in a reuse initiative program. It is calculated by adding the total size of the shipped software by a team or organization (SSI) to what the team or organization is reusing from the reuse library (RSI) and what was produced by that team or organization and is being reused by others (SIRBO). The result of this sum is then divided by SSI. The resulting index is a measure of the reuse effectiveness of that team or organization: an RVA=1 means there is no reuse activity involved at all, since RSI+SIRBO=0, while an RVA=2, for instance, means that the team or organization is twice as effective from a reuse perspective.

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

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

4.2.1 Software Reuse Impacts

Empirical studies, in both industry and academy, aiming to assess the relation of software reuse with different quality and cost metrics have been reported in the literature [Lim, 1994] [Henry & Faller, 1995] [Devanbu et al., 1996] [Basili et al., 1996] [Frakes & Succi, 2001]. All of the reported studies dealt with a very limited number of projects, what made their results inconclusive, but the general notion that software reuse and software quality are intrinsically related held true for all cases, while the inverse relation between software reuse and development cost failed to hold for some of the studies. Table 4.2 summarizes the measurable impacts of software reuse.

<table>
<thead>
<tr>
<th>Aspect</th>
<th>Measurable Impacts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quality</td>
<td>Error density</td>
</tr>
<tr>
<td></td>
<td>Fault density</td>
</tr>
<tr>
<td></td>
<td>Ratio of major errors to total faults</td>
</tr>
<tr>
<td></td>
<td>Rework effort</td>
</tr>
<tr>
<td></td>
<td>Module deltas</td>
</tr>
<tr>
<td></td>
<td>Developers perception</td>
</tr>
<tr>
<td>Productivity</td>
<td>Lines of code per effort</td>
</tr>
<tr>
<td>Time-to-Market</td>
<td>Development cycle time</td>
</tr>
</tbody>
</table>

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

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

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

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

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

The SORM are divided into two main categories: the amount of reuse metrics and the reusability assessment metrics. The former target assessing the reuse of existing items, while the later aim at assessing, based on a set of quality attributes, how reusable items are. Table 4.3 summarizes the main amount of reuse metrics defined so far.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reuse Percent (RP)</td>
<td>Ratio of the number of reused lines of code to the total number of lines of code [Poulin &amp; Caruso, 1993].</td>
</tr>
<tr>
<td>Reuse Level (RL)</td>
<td>Ratio of the number of reused items to the total number of items [Frakes &amp; Terry, 1996].</td>
</tr>
<tr>
<td>Reuse Frequency (RF)</td>
<td>Ratio of the references to reused items to the total number of references [Frakes &amp; Terry, 1996].</td>
</tr>
<tr>
<td>Reuse Size and Frequency (RSF)</td>
<td>Similar to Reuse Frequency, but also considers the size of the items in number of lines of code [Devanbu et al., 1996].</td>
</tr>
<tr>
<td>Reuse Ratio (RR)</td>
<td>Similar to Reuse Percent, but also considers partially changed items as reused [Devanbu et al., 1996].</td>
</tr>
<tr>
<td>Reuse Density (RD)</td>
<td>Ratio of the number of reused parts to the total number of lines of code [Curry et al., 1999].</td>
</tr>
</tbody>
</table>

For the explanation of the different amount of reuse metrics, consider the example application (SampleApp) with the structure depicted in Figure 4.2. Each part (square) corresponds to an item (a file, for example) that has a name and a size in number of lines of code (LOC) and the arrows correspond to references between the items (method calls, for example). Solid arrows are references to internal items and dashed arrows are references to external items. The application is composed of internal, or new, parts (1, 2, 3 and 4), the white squares, and external, or reused,
4.3. SOFTWARE STRUCTURE ORIENTED METRICS

Before proceeding to the reuse metrics calculations, some basic observable data must be collected. The reused code size (ReusedSize) is:

\[ \text{ReusedSize} = \sum \text{LOC}(\text{reused}) = \sum_{i=5}^{6} \text{LOC}(\text{part}_i) = 150 \]

Where \( \text{LOC}(\text{part}_i) \) if the number of lines of code of \( \text{pat}_i \). The new code size (NewSize) is:

\[ \text{NewSize} = \sum \text{LOC}(\text{newparts}) = \sum_{i=1}^{4} \text{LOC}(\text{part}_i) = 120 \]

Consequently, the application size (Size) is 270. The total count of parts (Count) in SampleApp is 5 and the total count of reused parts (ReusedCount) is 2. The total number of references (Refs) is 6 and the number of references to reused parts (ResuedRefs) is 3. The expanded size (ExpandedSize) of SampleApp is:
CHAPTER 4. REUSE METRICS

\[
\text{ExpandedSize} = \sum_{i=1}^{6} \text{LOC}(part_i) \times \text{refs}(part_i) = 360
\]

Where \( \text{refs}(part_i) \) is the number of references to \( part_i \).

Once the basic data is collected, the reuse metrics can be calculated. \textit{Reuse Percent (RP)} is the most basic reuse metric, used in current economics oriented reuse metrics. It is defined as the ratio of the number of reused lines of code to the total number of lines of code. The \( \text{RP} \) for SampleApp is calculated as follows:

\[
\text{RP(SampleApp)} = \frac{\text{ReusedSize}}{\text{NewSize} + \text{ReusedSize}} \times 100 = 55.6\%
\]

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

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

\[
\text{RL(SampleApp)} = \frac{\text{ReusedCount}}{\text{Count}} = 0.33
\]

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

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

\[
\text{RF(SampleApp)} = \frac{\text{ReusedRefs}}{\text{Refs}} = 0.5
\]

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

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

\[
RSF(SampleApp) = \frac{ExpandedSize - Size}{ExpandedSize} = 0.67
\]

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

*Reuse Ratio (RR)* is similar to \( RP \), but besides *black-box* (verbatim) reuse, it
also considers *white-box* reuse: items changed to a certain degree are also considered
as reused. Additionally, this metric is calculate as a ratio and not a percentage. For
*SampleApp*, considering that all reused parts are reused verbatim, \( RR \) is equal to
\( RP \).

*Reuse Density (RD)* is the ratio of the number of reused parts to the size
of the system in number of lines of code. As in \( RF \) and \( RL \), this metric may be
divided into *Internal RD (IRD)*, *External RD (ERD)* and *Total RD (TRD)*. For
*SampleApp*, the \( RD \) value is calculated as follows:

\[
RD(SampleApp) = \frac{ReusedCount}{Size} = 0.007
\]

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

It is important to stress that no single metric is able to depict precisely how
a system or organization is in terms of reuse and perfectly reflect its impacts. A
judicious use of a combination of metrics and observations must be conducted in
order to accurately estimate the actual consequences of a software reuse program.
Table 4.4 summarizes the metrics values for *SampleApp*. From this table, it is clear
that each metric represents certain aspects of the application. A thorough analysis of such metrics is necessary for a better understanding of the reuse reality of the application, taking the context of the application into account.

Table 4.4: SampleApp reuse metrics values

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>RP</td>
<td>55.6%</td>
</tr>
<tr>
<td>RL</td>
<td>0.33</td>
</tr>
<tr>
<td>RF</td>
<td>0.5</td>
</tr>
<tr>
<td>RSF</td>
<td>0.67</td>
</tr>
<tr>
<td>RR</td>
<td>0.56</td>
</tr>
<tr>
<td>RD</td>
<td>0.007</td>
</tr>
</tbody>
</table>

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

4.3.1 Reusability Assessment

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

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

4.4 Reuse Repository Metrics

Reuse repositories play an important role in reuse programs since they act as a convergence point of all reuse efforts, related to reusable parts production or consumption. The efficiency of reuse repositories in aspects such as availability and quality of
4.4. REUSE REPOSITORY METRICS

search results, for example, may be a determinant factor for a better reuse activity and greater positive impacts on the quality and the cost of the produced software.

The main reuse repository efficiency measurements are: cost to find adequate reusable parts, quality of contained reusable parts, number of successful reuse cases of contained parts, availability of the repository [Frakes & Terry, 1996] and the total value of a reuse repository (RSL ROI) [Poulin, 1996].

The cost to find adequate reusable parts is intrinsically related to the quality and performance of the search and browsing mechanisms provided by the repository. The quality of contained reusable parts depends on the mechanisms a repository has in place, such as certification [Alvaro et al., 2005], to assure the contained reusable parts meet some predefined criteria to guarantee their quality. The number of successful reuse cases of contained parts depends on the first two measurements and implies an additional traceability mechanism between reusable parts and reusing systems. The availability of the repository is related to the stability of the system and is an important aspect to assure the repository is available to developers when needed, otherwise reuse would be inhibited by the lack of availability of the repository system.

The total value of a reuse repository (RSL ROI) is calculated using SIRBO, defined in Section 4.2. The value of the total amount of code extracted and reused from a repository can be calculated using the RCA metric also presented in Section 4.2.

However, in order to have a more accurate estimate of the return of investment of a reuse repository, one has to also take into account the costs for maintaining the repository: licenses, effort of the staff involved in the maintenance of the repository and hardware required to host the repository, among others. The repository ROI is the calculated as the difference between RSL ROI and the total repository maintenance costs. It is similar to the Organizational ROI metric, presented in Section 4.2.

4.4.1 Reuse Repository Adoption Strategy

A common mistake that organizations make when creating a central reuse repository is known as the “three RSL phases” problem [Poulin, 1995]. In the first phase, when the repository is built or bought by the organization, it is empty or has very few parts and therefore developers do not find it attractive to search the repository.

The organization then reacts to this problem in the second phase by launching incentive programs for contributions to the repository. The result is a repository full of parts of questionable quality, making the repository still unattractive to de-
developers.

Once more, the organization reacts to the problem in the third phase by establishing strict certification criteria to the parts being submitted to the repository. The consequence is a repository with consistent, well tested and well documented parts, but developers have difficulties in finding software that meets their needs.

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

4.5 Summary

This Chapter presented the existing reuse metrics in detail and defined the focus for the metrics to be used in the environment toward software structure and repository metrics. The description of the adopted metrics is presented in Section 5.3.

With all underlying concepts properly described, the proposed solution is presented in the next Chapter along with a discussion about the decisions taken for the definition of the solution.
Chapter 5

ADMIREE Environment

Based on the result of the research on the software reuse, information retrieval and reuse metrics fields, presented in Chapters 2, 3 and 4, a set of requirements for an integrated reuse environment have been defined for the proposed solution and are initially presented in this Chapter. The focus of the Asset Development Metrics-based Integrated Reuse Environment - ADMIREE - is on assisting developers in achieving their tasks more efficiently by reusing existing assets, while providing managers with mechanisms for monitoring the reuse activity achieved by developers.

The technologies used to provide such environment are then discussed and a new reuse metric that assesses reuse from a similarity perspective between a set of artifacts and a reuse repository is proposed. The architecture of the system is then defined and finally the details of the implementation initially provided are discussed.

The discussion is organized as follows: Section 5.1 describes the requirements for the ADMIREE environment and Section 5.2 presents the underlying technologies of the environment. Section 5.3 details the proposed metric for the solution and Section 5.4 presents its architecture. Section 5.5 describes the initial set of functionalities provided, based on the proposed architecture. Finally, Section 5.6 summarizes the ADMIREE environment.

5.1 Requirements

The major shortcoming of existing solutions is that none of them addresses reuse issues, in the particular context of this work related to reuse metrics and reusable assets retrieval as discussed in Chapters 2 and 3, from both individual and organizational points of view in a systematic fashion. Moreover, pragmatic issues that
arise in corporate environments, such as distribution, scalability, security and man-
agement of the reuse environment are completely neglected, making a systematic
adoption of such environments in large organizations impractical.

The requirements proposed for ADMIRE are based on the authors’ experience
on software development and reuse, RiSE staff discussions, and on the literature
presenting requirements for effective reuse environments, search engines and reuse
repositories [Lucredio et al., 2004] [Garcia et al., 2006b].

Metrics extraction activities focus on tackling the reuse discipline from an
organizational perspective, aiming at answering questions like [Poulin, 1997]: “How
much is being reused by development teams?” and “How much money/effort/time
is being saved or spent because of reuse?”. They ought to be incorporated in the
development process, so there must be tools to automate this extraction. There are
few proposed reuse metrics, which were presented in Chapter 4, that have an actual
extraction tool implementation associated. The few ones that do have such tools
do not provide hooks for a systematic reuse metrics extraction. They are usually
presented as stand-alone, single-shot tools.

The problem with this approach is that incorporating reuse metrics extraction
to the development process requires an additional effort and discipline that can act
as inhibiting factors for a systematic adoption by development teams.

Information retrieval strategies, presented in Chapter 3 on the other hand, are
mainly concerned in handling reuse issues from the individual perspective by making
it easier for developers to find relevant information to be reused. Some solutions
go even further by taking a proactive approach and automatically detecting and
delivering Java components that might be reusable in the context of the developers’
tasks [Ye, 2001]. Still, these solutions treat information retrieval as an isolated,
although important, entity in the software reuse universe.

In this context, a set of requirements for what is envisioned as an Asset
Development Metrics-based Integrated Reuse Environment (ADMIRE)
are defined in the following Subsections.

5.1.1 Extensibility

Software development involves a large number of steps and different intermediate
types of asset are produced along the path. There are usually multiple alternative
formats to build each type of asset and on top of that, the information contained in
the assets may be encoded in different languages.

\[^1\text{http://java.sun.com}\]
5.1. REQUIREMENTS

Examples of asset types include use case specifications, documentation and programming units. A programming unit, in its turn, may be defined in a COBOL program, a C file, a Java class or in any of the several existing programming languages. To make the number of alternatives even bigger, the elements in a Java class (e.g. class and method names), for instance, can be encoded in English or Portuguese.

Since reuse depends on locating, understanding and incorporating concepts into newly developed artifacts, issues related to the type, format and language of the assets must be addressed.

For this reason, the proposed solution must be generic enough to accommodate different asset types, formats and languages. Hooks must be provided in order to allow the addition of new capabilities to the environment. This is a high priority requirement.

5.1.2 High Precision

Information retrieval performance is measured in terms of the achieved precision and recall (as discussed in Chapter 3) and the proposed environment heavily depends on information retrieval strategies to (1) assess the reuse potential of a set of assets given a legacy asset repository and (2) help keeping a high level of reuse activity by actively delivering reuse candidates for the user tasks.

Since large organizations tend to have a huge amount of legacy assets available, a poorly designed solution would lead to a vast amount of irrelevant information being delivered to users. As a result, users would either have their productivity affected by such information flood or would disregard the environment, making it useless [Zimmermann et al., 2004].

The ultimate goal is to keep the presented results as accurate as possible, even at the expense of missing some relevant information to the user task. This is a high priority requirement.

5.1.3 Ubiquity

No attempt to reuse is the number one software reuse failure mode [Frakes & Fox, 1995] and that is mainly caused by the lack of knowledge of existing assets that could be reused. A reuse within development approach [Ye, 2001] must be employed, making the cost of finding reusable work products as low as possible.

Thus, the proposed solution must provide programmers with a comprehensive set of tools that smoothly merge with existing development environments and tools.
in order to minimize the effort for achieving higher levels of reuse activity. Although important, this is not a high priority requirement.

5.1.4 Access Control

The legacy assets available in a repository usually have different sources and depending on these sources, the assets may be accessed only by developers on certain groups inside or outside organizations. For instance, a reusable component developed in a specific project for a client may only be accessed by other projects for the same client depending on the type of licensing adopted or due to internal restrictions enforced by organizations.

Therefore, the information retrieval mechanism used in the environment must address assets access control in order to avoid such inter-project licensing or restriction problems.

Different approaches may be followed to accomplish the required access control policy: search filtering and content blocking are the most commons ones. The search filtering restricts the search space to assets that are accessible to the user performing the query while the content blocking approach allows searches on the entire index, but blocks contents retrieval if the user does not have access to the required asset.

Information retrieval mechanisms that favor high precision should adopt the former approach if an unaccessible relevant asset is considered as irrelevant in the context of the user, since it is not possible to reuse that asset. This is a low priority requirement, since it can be implemented in future versions of the environment without impacting in the main functionality of the initial version.

5.1.5 Scalability

Although performance is not a highest priority requirement for the environment, the potential amount of operations performed for an active information delivery for a single user pushes for a solution that properly scales according to the number of users. Large organizations usually have hundreds or thousands of users performing development activities concurrently.

As discussed in Section 3.4, an active information delivery approach has a potential side effect of changing the course of action of the user according to the task at hand and the information delivered. The time when the information is delivered is of paramount importance for this to properly occur. Therefore, the information retrieval mechanism must yield results in a reasonable time regardless of the number of concurrent users and the activity load performed by them. This is
an important requirement, since it can be difficult to add this characteristic to the environment once it is fully operational if not planned upfront.

### 5.1.6 Dynamicity

The cost for developing reusable assets usually prevents organizations from adopting a systematic reuse program. Moreover, the incentives in place for projects teams diverge from the creation of reusable assets [Fichman & Kemerer, 2001]. Their main focus is on the delivery of the project’s specific products on time and the overhead for making such products reusable is usually neither affordable nor desirable.

To overcome this major inhibitor, the proposed solution must take an incremental approach into making assets more reusable as new opportunities for their usage are presented. It must be able to leverage, in some extent, the assets that have not been specifically designed or developed with reuse in mind, promoting a greater integration among development teams and consequently a greater reuse activity in organizations. This is a high priority requirement.

### 5.1.7 Continuous Metrics Extraction

From the organizational perspective, the reuse activity must be systematically monitored so the impacts of reuse over other development aspects, such as quality and cost, can be assessed and deviations can be timely detected and handled.

For this reason, automated metrics extraction tools must be provided by the solution in consonance with existing continuous integration practices [Clark, 2004]. This is a high priority requirement.

### 5.1.8 Feedback Support

Users experience is maximized when they are able to shape the environment behavior according to their knowledge and the context of their tasks [Ye, 2001]. One example of the maximization of the reuse activity is the case when the information retrieval mechanism gives higher priority to assets that have been successfully reused by other programmers when delivering reusable candidates to a specific programming task.

An integrated reuse environment must provide users with a feedback mechanism to incrementally improve the quality of the information retrieval and metrics extraction functionalities. This requirement can be fully implemented in future versions of the environment without impacting its main functionalities in the initial version.
5.1.9 Platform Independence

Organizations usually have heterogeneous development platforms and, for that rea-
son, an integrated reuse environment must seamlessly integrate with all existing
configurations in order to maximize its user base and consequently provide more
effective results.

The implementation of the environment functionalities must be based on tech-
nologies that are easily portable across existing platforms. This is a high priority
requirement.

5.1.10 Summary of Requirements

The proposed solution must fulfill a set of requirements to provide users with an
integrated reuse environment. Pragmatic considerations must be taken into account
in conjunction with state-of-the-art, innovative approaches in order to maximize the
reuse activity in organizations regardless of their size. The set of requirements is
summarized in Table [5.1].

From the requirements listed in Table [5.1] it is evident that the ultimate goal is
to provide users with a comprehensive set of functionalities that focus on facilitating
the reuse activity from the individual perspective and the reuse management from
the organizational perspective. Next Section discusses the underlying technologies
that support the environment.

5.2 Technologies

Since reuse is of great concern to this project, the main environment design principle
is to leverage existing technologies in order to provide users with the most sophis-
ticated mechanisms, while preserving pragmatic considerations such as scalability
and ubiquity in a reasonable time frame.

This Section briefly discusses the choices for the underlying technologies and
mechanisms adopted during the environment construction and the initial set of
supported asset types, formats, languages and development tools.

5.2.1 Information Retrieval Library

From the approaches to information retrieval, discussed in Chapter [3], there are
good quality open source libraries available, mostly on the Vector Space Model,
Table 5.1: Integrated reuse environment requirements

<table>
<thead>
<tr>
<th>#</th>
<th>Requirement</th>
<th>Priority</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Extensibility</td>
<td>High</td>
<td>The environment must be able to handle different asset types from various sources in different formats and in different languages.</td>
</tr>
<tr>
<td>2</td>
<td>High Precision</td>
<td>High</td>
<td>The information retrieval and metrics extraction mechanisms must minimize false positives (high precision) to prevent information flooding, even at the expense of missing relevant information (low recall).</td>
</tr>
<tr>
<td>3</td>
<td>Ubiquity</td>
<td>Medium</td>
<td>The provided set of tools must smoothly integrate with existing development environments in order to facilitate the access to the legacy assets, which can potentially improve the reuse activity.</td>
</tr>
<tr>
<td>4</td>
<td>Access Control</td>
<td>Low</td>
<td>The information retrieval and metrics extraction mechanisms must handle sensitive information that is available only for a subset of users of the environment.</td>
</tr>
<tr>
<td>5</td>
<td>Scalability</td>
<td>Medium</td>
<td>The environment must scale according to the number of users. Although performance is not a high priority requirement, user experience is augmented with timely responses from the environment.</td>
</tr>
<tr>
<td>6</td>
<td>Dynamicity</td>
<td>High</td>
<td>Partially reusable assets must be leveraged in some extent by the solution in order to promote a greater reuse activity in organizations.</td>
</tr>
<tr>
<td>7</td>
<td>Continuous Metrics Extraction</td>
<td>High</td>
<td>Metrics must be continuously monitored so deviations are timely detected and handled. The environment must provide users with tools in consonance with existing continuous integration practices.</td>
</tr>
<tr>
<td>8</td>
<td>Feedback Support</td>
<td>Low</td>
<td>The environment must provide users with feedback mechanisms in order to shape the behavior of the system according to the users knowledge and the context of the current task.</td>
</tr>
<tr>
<td>9</td>
<td>Platform Independence</td>
<td>High</td>
<td>In order to maximize its usage, the environment must not be tied to any particular platform, programming language or development tool.</td>
</tr>
</tbody>
</table>
Boolean and Probabilistic categories. From the various available open source information retrieval libraries, the most prominent ones are Xapian and Lucene [Gospodnetic & Hatcher, 2004].

The Xapian project[^3] is a C++ class library based on the Probabilistic Model that provides bindings to several programming languages, such as Java and C#. It provides query parsers for human-friendly search syntax and porter stemmers to reduce words to their stems [Rijsbergen et al., 1980].

The Xapian model defines a Document class that represents a document in the Xapian database. This class holds data, values and terms of the documents. Additionally, queries can be defined with the Query class, which provides a set of boolean and proximity operators for complex queries formulation.

The Lucene project[^4] is a Java class library with ports to other programming languages, such as C++ and C#, that provides Boolean and Vector Space Model capabilities. Its model is highly extensible and well designed. Lucene has a set of features comparable to Xapian and although no formal comparison has been performed, both are regarded as high quality information retrieval libraries, being widely adopted in TREC[^5] conferences.

Similarly to the Xapian model, the Lucene model defines a Document interface that is used for information indexing. A Document is composed of a set of custom defined fields. The queries may be performed on a single field or on multiple fields. There is a fixed set of predefined field types that determine the way the indexing and querying engines will handle them. Complex information retrieval strategies may be created on top of this flexible model.

### 5.2.2 Content Parsing

Although one of the design principles of the environment is extensibility (requirement #1), the initial set of supported artifact formats is limited to Java source classes. Working at the code level was the more natural for an initial environment implementation, since another strong requirement of the solution is to be able to handle partially reusable artifacts (requirement #6), probably with inexistent or incomplete documentation. Furthermore, usually the only reliable source of information is the source code, especially for legacy systems during maintenance phase [Souza & Moreira, 1998] [Gold & Mohan, 2003].

Source code parsers take files that match specific grammars as input and

[^3]: http://www.xapian.org
[^4]: http://lucene.apache.org
[^5]: http://trec.nist.gov
5.2. TECHNOLOGIES

generate their abstract representation, called Abstract Syntax Tree (AST) [Aho et al., 1986]. The main parser types are Left-to-right, Leftmost derivation (LL) and Left-to-right, Rightmost derivation (LR). Parser generators take grammar definition files as input and generate parsers for the specified grammars.

Two widely used parser generators are ANTLR [Parr & Quong, 1995] and JavaCC [7], both LL parsers. There are several grammars for most of the modern programming languages available for both of these parser generators.

Other types of artifacts besides source code may require different tools for a proper content parsing. One example of such different strategies is for parsing rich format documents. For instance, Portable Document Format (PDF) documents may be parsed by a number of open-source libraries that extract the textual information contained in such documents.

5.2.3 Integrated Development Environment

The support for the reuse activities must be fully integrated into the programmers work environment in order to fulfill the ubiquity (requirement #3) requirement. Integrated Development Environments (IDEs) provide users with facilities to writing, compiling, running and debugging source code, among many other features, inside one single environment.

Currently, the most used open source IDEs are Eclipse [8] and NetBeans [9]. Both are elegant, modular and highly extensible systems. Eclipse is based on the OSGi Framework [10] and its official release supports different programming languages, including Java, COBOL and C/C++. For being widely used and natively supporting a wide range of programming languages, Eclipse was chosen to be initially integrated with the provided framework.

5.2.4 Continuous Integration Tool

Continuous integration tools rely on build tools to perform the applications’ build process. Unquestionably, on the Java world, the most used build tool is Ant [11]. It provides a set of standard tasks that may be combined to perform sophisticated

---

6 http://www.antlr.org
7 https://javacc.dev.java.net
8 http://www.eclipse.org
9 http://www.netbeans.org
10 http://www.osgi.org
11 http://ant.apache.org
build processes. Custom defined tasks may also be plugged to the engine to fit specific needs.

For the aforementioned reasons, the Ant build tool will be used as the integration point for the metrics extraction functionality provided by the environment. A custom Ant task provides the necessary hook for a continuous metric extraction.

5.3 Proposed Metric

The reuse metrics presented in Chapter 4 have in common the fact that they basically aim at assessing, although in different ways, how much was reused during the construction of a software product (usually an application). For this reason, they can all be seen as realized reuse metrics.

While this is a fundamental aspect to be considered when assessing the reuse activity, there is a critical detail that is lacking on all proposed reuse metrics so far: the reuse potential of a product relative to a repository of legacy assets. That is, given a repository of (semi) reusable assets, the following question must be answered: “How much could be reused when building this new application?”.

Figure 5.1 illustrates this simple concept in a scenario. From this scenario it is clear that merely comparing the achieved reuse level of Project A and Project B will be biased towards Project A, since nothing from the repository fits in the parts of Project B. Thus, it can be said that Project B does not have what to reuse from the repository. A more complete assessment would have to involve both the potential and achieved reuse relative to a reuse repository. This combination describes the actual reuse activity in projects more precisely.

Once the necessity of definition of a reuse potential metric is agreed upon, the remaining issue is how to perform the calculation of such metric and for that, some information retrieval concepts must be employed.

The artifacts produced in a specific project belong to the query space (A1 through A3 and B1 through B3), while the assets available in a repository belong to the search space (R1 through R6). The proposed metric is defined in terms of the set of queries extracted from the query space and, from this set, the number of successful queries against the search space.

The rationale for this metric is that in a perfectly reuse-oriented development, before starting the construction of any artifact, the developers would perform queries to a reference repository in order to find out if there is a previously developed asset that may be reused in the context of the current task at hand. The problem with this scenario is that it requires a great discipline from developers and demands an
5.3. PROPOSED METRIC

Figure 5.1: Reuse realization versus reuse potential

overhead that impacts negatively to the overall productivity.

For a more in-depth explanation, the reuse repository must be considered as a hierarchical aggregation of reusable assets, where assets are further refined into lower level assets. In this context, there are multiple possible levels of granularity for assets reuse, although some levels may be more suitable for reuse than others.

The higher up on that tree, the harder to reuse and at the same time the greater benefits from reuse. This type of reuse is usually called vertical reuse. On the flip side, horizontal reuse, for lower nodes at the tree, is usually easier to achieve but it yields smaller benefits when considered individually. Both types of reuse, however, are extremely desirable whenever possible.

Reuse opportunities may be related to an individual asset or to a set of interrelated assets. Back to the Java program example, a program is composed of modules or components, which are composed of Java packages. Java packages are an aggregation of related Java classes that are an aggregation of methods and methods, on their turn, are an aggregation of statements. A single class, for instance, may be reused as an isolated asset or a set of related classes, usually following a design pattern [Gamma et al., 1994], may be reused in a different context. Figure 5.2 better illustrates the possible individual and collective reuse types within a Java program.

The proposed metric builds on the concepts exposed so far and depends on
some additional definitions:

- Given a collection of artifacts (cart) (e.g. a Java program), the query set (qs) of that collection is composed of the minimal set of queries that are directly derivable from that collection and that represents the contents of that collection. The query set is calculated with a query formulation function (form(cart)). The query formulation function may work on any level of the hierarchy and may derive queries for individual or interrelated assets. For instance, the query set from a Java module could contain the name of the packages, classes, methods or any arbitrary combination of those names;

- Given a reuse repository (rr) and a query (q), the result set of the execution of that query on that reuse repository (rs(q,rr)) is composed of potential reuse candidates (cand) that match the query and are indexed by the reuse repository. Each candidate has its associated relevance or ranking (rank(cand)) for the context of the query. For the Java module context, the result set could contain a list of methods that match the query; and

- Given a reuse repository (rr) and a query set (qs), the successful query set (sqs) is the subset of qs where the result of a result set evaluation function (eval(rs)) is greater than a specified threshold (thr). The evaluation function
5.3. **PROPOSED METRIC**

may be based on the *ranking* of the *candidates* contained in the *result set*, considering just the most relevant result or any arbitrary combination of the yielded results.

The *reuse potential* \((rp(cart, rr))\) of a *collection of artifacts* given a *reuse repository* is then defined as the normalized value in **Equation 5.1**

\[
rp(cart, rr) = \frac{\#sqs(cart, rr)}{\#qs(cart)}
\]

The *reuse potential* may be seen as a measure of *similarity* between a collection of artifacts and the artifacts indexed by a reuse repository. Reuse opportunities are greater when the similarity measure yields greater values. Figure 5.3 illustrates the reuse potential extraction process given a collection of artifacts and a reuse repository.

Figure 5.3: Outline of the Reuse Potential extraction process

From this example extraction process, the resulting *reuse potential* is calculated, according to **Equation 5.1**, as the ratio of the *cardinality of the success query set* to the *cardinality of the query set*, as follows:

\[
rp(cart, rr) = \frac{\#sqs(cart, rr)}{\#qs(cart)} = \frac{3}{6} = 0.5
\]

For a more concrete example, consider the following Java class:
public class Bank {
    public float checkAccountBalance(Account account) {
        float balance = 0;
        ...
        return balance;
    }

    public void withdraw(Account account, float value) {
        ...
    }
}

The query set could be extracted from the Bank class:

qs = {q1, q2, q3}
q1: Bank
q2: checkAccountBalance
q3: withdraw

After the execution of the extracted queries against a reuse repository, the following result sets could be retrieved (for the sake of simplicity, each result only contains the name and the relevance of the reuse candidate):

rs(q1) = {(SwissBank, 0.65), (FinancialInstitution, 0.34)}
rs(q2) = {}
rs(q3) = {(getMoney, 0.55), (getCash, 0.55), (removeFromAccount, 0.55)}

The evaluation on the result sets could consider the head of each result set and compare its relevance to a specified threshold or could consider a combination of the top relevant results. For example, if the threshold is 0.60 and just the head is considered, the only successful query would be q1. If, on the other hand the top results are considered, query q3 can be included in the successful query set because the top three results are close to the specified threshold:

sqs = {q1, q3}

And the resulting reuse potential would be $\frac{2}{3} = 0.67$.

A binary relation can be derived from the reuse potential metric in order to establish some important properties that such metric must hold. The reuse
5.3. PROPOSED METRIC

Potential relation is then defined in terms of a threshold (thr) as the relation between a collection of artifacts (cart) and the artifacts indexed in the reuse repository (rr) such that the reuse potential value between cart and rr is greater than the established threshold. Equation [5.2] describes the reuse potential relation.

\[
5.2 \quad rpr_{th}(cart, rr) \Rightarrow rp(cart, rr) \geq thr
\]

Binary relations must hold three important properties: symmetry, reflexivity and transitivity. Symmetry determines that an element is related to itself. In the context of this work this means that if a collection of artifacts is indexed in a reuse repository and the reuse potential of that same collection is calculated against that repository, the resulting value will be greater or equal to the established threshold. This is illustrated in Equation [5.3].

\[
5.3 \quad rp(cart, cart) \geq thr \Rightarrow rpr_{th}(cart, cart)
\]

The reflexivity property determines that if a collection of artifacts (cart\(_1\)) is related to another collection of artifacts (cart\(_2\)), as defined in Equation [5.2], then cart\(_2\) is also related to cart\(_1\), as shown in Equation [5.4].

\[
5.4 \quad rpr_{th}(cart_1, cart_2) \Rightarrow rpr_{th}(cart_2, cart_1)
\]

Finally, the transitivity property states that if a collection of artifacts (cart\(_1\)) is related to a second collection of artifacts (cart\(_2\)) and cart\(_2\) is related to a third collection of artifacts (cart\(_3\)), then cart\(_1\) and cart\(_3\) are also related, as Equation [5.5] states.

\[
5.5 \quad rpr_{th}(cart_1, cart_2) \land rpr_{th}(cart_2, cart_3) \Rightarrow rpr_{th}(cart_1, cart_3)
\]

All the underlying concepts to the reuse potential metric and its derived reuse potential relation are defined in a generic fashion in order to preserve the extensibility of the environment. The query formulation, query execution and result evaluation functions may be instantiated according to a number of factors, like the type of the artifacts in the working set, for instance.

Should the resulting reuse potential metric hold the aforementioned properties, important optimizations can be performed during the metric extraction process.
from a large set of artifacts collections by avoiding part of the metric extraction steps and inferring the reuse potential values from the extracted ones in conjunction with the aforementioned properties. Section 6.3 further discusses these optimization aspects.

The reuse potential metric plays an important role to the proposed environment by serving as a consolidation of the individual perspectives of a development team. Its main focus is to grasp the reuse activity from the organizational perspective in conjunction with other reuse metrics detailed in Chapter 4.

It is important to stress that the proposed metric does not have a direct relation with other software development aspects such as quality and cost, as discussed in Section 4.2.1 for other reuse metric categories, since it only assesses the amount of reuse that can be achieved without concerning about the actual achieved amount of reuse.

### 5.4 Environment Architecture

A general definition of the architecture of the ADMIRE environment is presented in this Section. The overall goal is to satisfy the set of requirements defined in Section 5.1 in a consistent way, providing a unified vision of what the environment looks like and how its internal components are combined in order to provide users with an integrated reuse environment. Although formal methods for deriving a system architecture from the requirements exist [Bass et al., 2003], such derivation process is out of the scope of this work.

The main phases of the reuse program that the environment should support are, basically: (1) legacy content retrieval, (2) repository indexing, (3) activity monitoring, (4) query formulation and execution, (5) results evaluation and presentation, (6) indexed contents retrieval, and (7) metrics extraction. For each of these phases there is a set of components that work in conjunction to provide the necessary functionality. Figure 5.4 depicts the overall environment architecture.

Each developer, normally, has a limited view of the working set. This working set window, named *Developer Working Set* (DWS), corresponds to the artifacts involved in the developers’ current task. The *manager* role, on the other hand, has a broader view and its responsibility is to monitor the development activities to ensure that the reuse opportunities across the whole working set are realized as much as possible. Metrics consist on important indicators of the reuse activity that assist on this monitoring.

The modules shown in Figure 5.4 perform the necessary operations for the phases of the environment. The dashed arrows represent planned operations that
Figure 5.4: ADMIRE, the architecture

are not implemented in the initial version of the environment (Retrieve, Write, Store, Get, Index and Read). Also, the Crawler and the Tracer modules, faded in the Figure, are not available in the initial set of functionalities.

5.4.1 Legacy Contents Retrieval

The legacy contents retrieval phase consists of incrementally and continuously monitoring the produced assets in an organization and making those assets available for indexing so they can be potentially reused in different contexts. An asset evaluation policy is necessary for determining whether a specific set of assets (from a project, for instance) is proper for future indexing.

This policy is responsible, from a very high level standpoint, for filtering low quality assets that would more likely either negatively impact the retrieval performance or somehow cause troubles if reused. Additionally, this policy is responsible for determining whether the index of previously indexed contents should be updated. This accounts for the dynamic nature of the indexed artifacts, as state in requirement #6.

The legacy contents retrieval module (Crawler) accesses the source of infor-
mation on the various existing version control system repositories, such as CVS \footnote{http://www.nongnu.org/cvs} and Subversion \footnote{http://subversion.tigris.org}, as well as any other source of information, such as file systems and databases, and making them available for the indexing and indexed contents retrieval phases. This is done so the actual source of information is decoupled from the environment once its contents are retrieved, eliminating the dependency to the original source and possible communication failures during search execution or content retrieval from end users. This approach is similar to the one adopted by the Google web search engine [Brin & Page, 1998].

5.4.2 Repository Indexing

Once the source of reusable information has been retrieved, the indexing phase takes place. During this phase, performed by the \textit{Indexer} module, the contents of the available artifacts are parsed, according to their type and format, and analyzed before being actually indexed.

The \textit{parsing} is responsible for understanding the types and formats of the available artifacts and extracting their contents to a normalized representation, while the \textit{analysis} is responsible for interpreting this normalized representation and determining whether it is relevant and therefore should be indexed.

Some manipulations of the contents may be performed during analysis in order to maximize the performance of the information retrieval mechanism by \textit{contracting} or \textit{expanding} the result set according to the goal of the mechanism, as discussed in Section 3.1. Examples of such manipulations include \textit{handling identifiers} [Michail & Notkin, 1999] [Jensen, 2004], \textit{reducing words to their stems} [Rijsbergen et al., 1980], converting words to their phonetic representation and \textit{synonyms handling} [Gospodnetic & Hatcher, 2004], among others.

5.4.3 Activity Monitoring

The ubiquity requirement (# 3) is satisfied by a set of tools and mechanisms that transparently aid the developer in finding and retrieving reusable assets that apply to their current tasks. The most prominent mechanism of this set is the activity monitoring, performed by the \textit{Listener} module. It interacts with existing integrated development environments as a listener in order to actively monitor and interpret the activities performed by the user, like adding methods to a Java class or creating a new COBOL program.
From this interpretation, the query formulation and execution agents are able to search the repository index and subsequently the results evaluation agent and the presentation module are able to interpret the search results and suggest reuse opportunities to the user. These three phases combined together consist on the active information delivery mechanism of the integrated reuse environment.

### 5.4.4 Query Formulation and Execution

The queries are formulated by the query formulation agent, contained in the Searcher module. The formulation is based on the contents of the artifacts being developed by the user. A similar approach to the repository indexing phase is performed in this phase, although the actual analysis performed may differ due to the distinct nature of development for and with reuse.

To clarify this distinction, a Java class as an example of artifact can be considered. A private method\(^{14}\) may be discarded during the analysis of the repository indexing phase, for being considered as an internal implementation detail of the class and thus not eligible for reuse opportunities. On the other hand, during the query formulation phase, a functionality that the user is trying to implement as a private method might be available as a reusable public method\(^{15}\), indexed in the repository. Thus, it makes sense to include the users’ private method during this phase.

For this reason, although the approach is similar, the actual mechanisms used by these two phases may differ depending on the particularities of the asset type and format.

### 5.4.5 Results Evaluation and Presentation

Once a search is performed and the results are retrieved, a detailed analysis of the results is necessary. The result evaluator agent, contained in the Searcher module, is responsible for detecting candidates that should not be presented to the user, based on the feedback provided from previous interactions or on information that is not available in the index and therefore cannot be taken into account during search time. One example situation of such filtering information is when the working artifact already references some of the candidates contained in the search result, making it pointless presenting them to the user.

Other type of information that is important during this phase and is not available in the index is the traceability data for each reusable artifact. This data, pro-

---

\(^{14}\)In Java, private methods are only accessible inside the declaring class.

\(^{15}\)Public methods, on the other hand, are accessible by any class.
vided by the *Tracer* module, consists on all clients of a given asset, that is, all assets that directly or indirectly reference that asset. This information may be used for *ranking* the results, giving a higher priority for assets that are used more frequently. The rationale is that assets that are used more often tend to be better quality assets, since they have been tested and debugged in different scenarios. This approach is inspired by the *PageRank* citation ranking [Page et al., 1998], used in the Google web search engine.

The remaining results from the analysis are then manipulated and finally presented to the user by the *Presenter* module. One example of such manipulations is *clustering the search results* [Osinski et al., 2004] for an augmented presentation experience. This module is responsible for determining how and when these candidates are presented to the user. Cognitive issues like the *level of intrusiveness* of the delivery must be taken into account when making these decisions [Ye, 2001].

### 5.4.6 Retrieval of Indexed Contents

The search results are presented to the user by the *Presenter* module and if the available, synthesized, information of one of the candidates contained in the results looks similar to what is really needed in that context, the next step is the *indexed contents retrieval* phase, performed by the *Searcher* module upon request from the *Presenter* module and initiated by the user. This phase is responsible for providing the system with user feedback (which assets were considered relevant to the user in his context) and retrieving the actual information source from the repository system (*Repository*).

As previously mentioned, the original source of information is not involved in this phase, so possible communication problems during content retrieval from the end user are avoided. Instead, the contents stored in the *Repository* (cached) during the legacy contents retrieval phase is returned to the user.

### 5.4.7 Metrics Extraction

Complementarily to the *active information delivery cycle* consisted of the previous phases, the *metrics extraction* phase takes a more general look at the produced artifacts from the organizational perspective. All the previous phases, except the *legacy contents retrieval* and *repository indexing* phases, are focused on reuse from the individual perspective, aiming at helping developers in achieving a higher reuse activity.

\[\text{http://www.google.com}\]
5.5. **OUTLINE OF IMPLEMENTATION**

This phase, performed by the Extractor module, is responsible for ensuring that given a set of artifacts being produced (the Working Set) and the available reuse repository system (the Repository), the development team extracted the most out of the repository when building the new set of artifacts. In other words, good reuse candidates presented by the system to the user have not been neglected.

It is important to note that negligence may not be the only cause for such misses, given the dynamic nature of the proposed environment. It is possible, for example, that good reuse candidates have been added to the repository between the end of development and the exact moment of the metric extraction. This situation, although possible, is very unlikely to occur if a continuous metric extraction discipline is being employed and even if it does happen, timely corrective actions may be taken.

For the reasons exposed so far, the natural conclusion is that finding actual good reuse candidates is a key factor for the success of the solution. For this reason, the precision of the retrieval mechanism must be prioritized over its recall in order to avoid undermining the environment credibility [Zimmermann et al., 2004].

As a final statement, it is important to stress that the mechanisms used during metrics extraction must be fully compatible with the mechanisms used in the previous phases to ensure that the extracted metrics reflect, in a consolidated manner, what the users had access to during development through the information delivery mechanisms. In other words, the metrics values yielded by the extraction tool must be proportional to the number of good reuse candidates presented to the user by the information delivery mechanism.

---

**5.5 Outline of Implementation**

As stated earlier in this Chapter, the initial implementation of the solution comprehends a subset of the features contemplated by the architecture of the environment, detailed in the previous Section. The main goal of the initial implementation is to provide a strong foundation with core components, where new components may be incrementally incorporated, improving the overall performance of the environment and making it suitable for a broader range of contexts.

Figure 5.5 depicts the initially provided integrated reuse environment components and the integrations with existing development and general usage tools. Design decisions and the internal details of the components are presented in the remainder of this Section.
5.5.1 Common Elements

The system architecture relies on a set of common elements, shared by the core components for the consistency between the extracted metrics and the information delivery mechanism. These elements are detailed in this Section.

Parser

The artifact parsing is managed by the parsing manager component that can be configured to associate file types to parsers. The initial association, restricted to Java source files, is performed by a parser generated by the ANTLR parser generator, described in Section 5.2. The Java grammar that was used is available in the official ANTLR distribution. Unrecognized (not associated) file types are ignored by the parsing manager and a specific type can only be associated to one parser.

Analyzer

The analyzer extracts the relevant information out of the generated AST (as described in Section 5.2.2) for later indexing or searching. The extracted information represents the asset hierarchical structure in an internal format, detailed in Appendix A. As with the parsing manager, the analysis manager can be configured to
associate asset types, anywhere in the asset hierarchy, to analyzers.

The analyzer initially provided extracts the name of the methods in a Java class. The structure of the method is analyzed in order to determine whether it is a relevant method in terms of reuse and thus should or not be extracted. This is done to avoid indexing or searching pure access methods, which usually consist on a large percentage of public methods of Java classes. Access methods' sole responsibility is to provide access to information encapsulated in objects, not consisting on valuable reusable assets. This filtering process yields a better search accuracy and smaller index sizes and constitutes an important contraction manipulation (defined in Section 3.1).

The extracted method names are then processed by separating the words contained in the method name using an approach similar to the one in [Jensen, 2004]. For instance, after this step the method named checkAccountBalance would be divided into the three terms: check, account and balance.

A set of stop words, detailed in Appendix B, is then checked in order to remove very common terms with a small discriminating power. The remaining terms are then stored into the internal representation of an asset of type 'method'. This constitutes another important contraction manipulation.

Since precision is of paramount importance for the environment, no expansion manipulation is performed during analysis in order to keep the number of retrieved results small.

5.5.2 Core Components

The basic functionality of the solution is provided by the set of core components described in this Section. These basic components are (1) Searcher; (2) Indexer and (3) Extractor. They all share a set of common elements described in Section 5.5.1.

Indexer

The indexer and the searcher components use the Lucene class library, described in Section 5.2. The platform independence provided by Java (although Xapian provides bindings to different languages, its core is platform-dependent, requiring some extent of porting effort), the extensibility of the model provided by Lucene and the vastly available documentation [Gospodnetic & Hatcher, 2004] and user support were key factors in deciding for this library as the core information retrieval mechanism used in the solution. Furthermore, the Lucene class library is used other RiSE\textsuperscript{17} projects,
such as Maracatu [Garcia et al., 2006].

The contents of the fields of the documents stored in the index are extracted from the internal representation, generated by the analyzer element.

**Searcher**

The searcher component is responsible for formulating and executing queries. The actual query formulation is delegated to the query formulator element, which in turn uses the same set of artifact parsers and analyzers that are used during indexing time. Like with the indexing component, the internal representation generated by the analyzer is then used during query formulation.

Since the volume of performed queries can be considerably large, especially for the active information delivery part, a query variation checker element is used to determine whether a new set of formulated queries is significantly different from the previously formulated set and therefore should be executed or not. The query manager element is responsible for caching the query sets and performing this check.

The search result ranking is performed on top of Lucene’s default relevance function by the ranking element that reads data from the tracer component to prioritize more widely used assets. The proposed model also contemplates a search result filtering mechanism based on an access control policy, although such policy is not implemented in the initial set of features of the environment.

**Extractor**

The extractor component works as a batch search component executor. All formulated queries for a given set of artifacts are executed and the results are analyzed by the search result analyzer element, that determines whether queries were successful or not. The reuse potential metric is then computed with the total number of executed queries and the number of successful queries, according to Equation 5.1.

The success of a query is determined by the ranking value of the most relevant document in the result set that is compared to a configurable threshold value. The rationale for only considering the most relevant document is that this tends to yield higher precision results, since documents with smaller ranking values are less likely to be relevant to the context of the query.
5.5. OUTLINE OF IMPLEMENTATION

5.5.3 Integrations

The environment must provide for integrations with existing development and general usage tools in order to transparently provide developers and managers with an integrated environment that maximizes the reuse activity throughout the organization.

The core implementation provides an extensible framework that supports integrations with a small effort from the solution side. The actual effort depends on the tool being integrated. The initial set of integrations includes a build tool, an IDE and an instant messenger. Each one of these integrations is described later in this Section. The goal is to demonstrate how the solution fits into tools and environments that are normally used on a daily basis by developers and managers.

**Build Tool**

An *Ant task* is provided to invoke the *Extractor* component, described in Section 5.5.2. Ant tasks are custom defined Java classes that are invoked by the Ant engine during build time whenever the task definition is invoked by an Ant build script.

This task can be incorporated to the set of tasks to be performed during build time in a project. The *reuse potential* metric can then be computed every time the system under development is built and presented to the user (the manager) in a generated report file, although this functionality is not initially implemented. Figure 5.6 shows the output generated by the Ant task during build time.

**IDE**

An *Eclipse plug-in* is provided to invoke the *searcher* component and present the search results in the editor window. The plug-in listens to code changes performed by the developer and, under the covers, determines when the *searcher* component should be invoked. An internal thread consumes a change event queue in order to decouple the query processing from the Eclipse’s event dispatching thread.

Once the query result set is available, the plug-in creates *Eclipse markers* that are displayed as small icons on the line of the asset (method, class etc.) that originated the query and a display text with the results from that query. Since the plug-in builds on top of Eclipse’s extensible framework, all features provided by the platform, such as the *message tab* that displays all active messages in a consolidate view and the *hierarchical element marking*, that automatically marks all ascendants of a marked file. Figure 5.7 illustrates all the features provided by the Eclipse plug-in with an actual Java sample class.
Figure 5.6: Ant *reuse potential* extraction task running inside the Eclipse IDE.

Figure 5.7: Eclipse plug-in features screen. (A) Hierarchical file markers, (B) Line markers and (C) Markers tab.
5.6. SUMMARY

The presented marker icon may have associated actions to be performed upon user request, but the initial implementation does not provide such functionality. Examples of usable actions include downloading the contents of the reusable asset and providing feedback about the asset to the environment.

Instant Messenger

A MSN Messenger robot is provided with the goal of demonstrating a non-conventional use of the integrated reuse environment. It was built on top of the JMML open-source Java library. Additional instant messaging integrations could be provided. For instance, the Jabber instant messaging platform could easily be integrated through the Smack open-source Java library.

The robot is responsible for signing in the MSN network, listening for instant messages that are treated as regular queries from the users, which must have the robot added to their contact lists, and replying to these messages with the query result sets. The management of the robot’s contact list (allowing or denying new contacts) is not initially implemented and must be performed manually.

The implementation of the MSN Messenger robot is similar to the implementation of the Eclipse plug-in. The additional concerns are regarding the size limit of the instant messages and the timing between subsequent messages, since the MSN network blocks messages that are sent with very small intervals to avoid intentional message flooding. A message sender element handles these concerns by breaking the result message into messages that fit into the size limit and enforcing a minimum delay between messages sent back to users. The reply messages may be formatted as hyperlinks for asset contents downloading, although this is not initially implemented. Figure 5.8 illustrates the MSN Messenger robot in action.

5.6 Summary

This Chapter presented the main aspects of the ADMIRE environment. The requirements for an integrated reuse environment and the proposed reuse potential metric were initially defined. The environment architecture was described and the set of technologies employed during its construction where discussed. Finally, the initial implementation’s internal components and set of integrations were described.

\[\text{http://messenger.msn.com}\]
\[\text{http://sourceforge.net/projects/pmlibs}\]
\[\text{http://www.jabber.org}\]
\[\text{http://www.jivesoftware.org/smack}\]
in detail with an analysis of the implementation decisions.

The next Chapter presents the experiments performed to evaluate the reuse potential metric extraction and consequently the information retrieval mechanism with a set of real Java programs.
Chapter 6

ADMIRE, the Evaluation

Once the ADMIRE environment has been described and its initial implementation detailed, some experiments should be performed to evaluate the precision of the provided functions used for the metric extraction process. In this sense, some projects have randomly been selected from the portfolio of a midsized software development organization\(^1\) for the experiments in order to ensure that the environment is suited for industrial contexts.

This Chapter is organized as follows: Section 6.1 presents the methodology of the experiments and the questions that must be answered by its results. The projects used during the experiments are described in Section 6.2, while Section 6.3 presents and analyzes the results of the experiments. Finally, Section 6.4 draws some conclusions on the findings of the evaluation.

6.1 Methodology

As stated in Chapter 5, the main goal of the ADMIRE environment is to serve as a unified solution for the increment of the reuse activity from both individual and organizational perspectives. In order to properly validate the proposed solution, the experiments to be performed must then reflect as closely as possible the actual type of context that is targeted by the environment.

For this reason, the approach of having empirical studies with tasks *specifically designed for the experiment* as followed in [Ye, 2001], [Jensen, 2004] and [Holmes, 2004], for example, do not consist on adequate evaluation strategies. Instead, the experiments must be performed on a set of *real software development*

\(^1\)There are approximately 800 collaborators working in this organization.
projects in order to properly validate the proposed environment.

Empirical studies on real projects under development would be impractical for several reasons in the context of this dissertation. For instance, time constraints, since real projects usually span across months and initial reluctance by development teams for adopting an experimental environment in real projects where deadlines must be met and risks must be avoided as much as possible. Additionally, this type of experiment usually requires that experiments are small scaled so they can be controlled and this scenario of small having small projects is not suitable for the goal of the evaluation of the environment.

Instead, the source code of already concluded projects, available in the legacy code repository of a midsized software development organization in Brazil and described in Section 6.2, will be used for the experiments. The downside of this approach, when compared to the former alternative, is that the impacts of the information delivery mechanism cannot be assessed, as performed by Ye in [Ye, 2001], because the artifacts under evaluation are already concluded and no intervention from developers occur during the evaluation. The results of Ye’s work, though, show that it is reasonable to assume that such impacts are important.

Additionally to the aforementioned considerations, the methodology used for the experiments had to be carefully planned, given the innovative approach taken by the proposed environment. The next Sections discuss the traditional approaches for evaluating reuse metrics and information retrieval solutions and the necessary adaptations for a proper evaluation of the ADMIRE environment.

6.1.1 Information Retrieval Evaluation

The usual approach for evaluating the performance of information retrieval systems is based in terms of the precision and recall metrics (as presented in Chapter 3) by utilizing a large dataset along with a set of queries and expected responses [Baeza-Yates & Ribeiro-Neto, 1999].

In the context of this work, such query and response sets are not easily identifiable due to the nature of the dataset, since the selected projects cover a broad range of domains unknown to the author. Moreover, there was neither enough documentation nor available time for an in depth understanding of the concepts involved in all selected projects and subsequent definition of a proper query/responses set.

---

2 C.E.S.A.R - Recife Center for Advanced Studies and Systems (http://www.cesar.org.br), has approximately 800 collaborators, congregates around a dozen startup companies and is preparing to obtain CMMI level 3.

3 This approach is followed by the Text REtrieval Conference (TREC) conferences.
6.1. METHODOLOGY

For this reason, an alternative approach, inspired by the work of Garcia et al. presented in [Garcia et al., 2006], will be adopted. According to this approach, a known project project is inserted in the repository for the experiments to serve as a reference for the evaluation of the results of the performed queries. In the context of this dissertation, the known project approach will be implemented by manually formulating the queries from the very resources of the selected projects. With this approach, the expected results for the query are exactly the assets contained in the known projects. This facilitates the evaluation of the information retrieval performance by non-experts. Figure 6.1 illustrates project A as the known project in a hypothetical experiment.

![Diagram](image)

Figure 6.1: The known project approach: queries are formulated from the contents of the very projects contained in the index

6.1.2 Reuse Metrics Evaluation

Evaluation of reuse metrics usually aim at assessing the impacts of reuse over development aspects, such as quality and cost, by comparing the achieved reuse levels with the data collected from projects (as presented in Section 4.2.1). This approach, however, is not suitable in the context of this evaluation for two reasons.

First, no reuse repository containing all the selected projects actually existed during the development phase of such projects. Although some informal communication may have occurred among the development teams their development efforts
were conducted isolated from each other, from an organizational standpoint.

Trying to establish a correlation between the reuse activity and different aspects of a project relies on the premise that projects that treated reuse on a more systematic way tend to have a better quality (i.e. less errors) and demand less effort [Basili et al., 1996], for example. The inexistence of a reuse repository and a systematic reuse strategy for the organization during development of the project invalidates this premise, since there was nothing to be reused from the perspective of the experiment.

Second, instead of trying to assess the amount of reuse of a system, the proposed metric is more concerned in assessing how much reuse can potentially be achieved given a set of artifacts and a reuse repository. Although these are interrelated concepts, assessing the reuse potential of a system does not necessarily lead to the actual amount of reuse achieved. These are complementary aspects of the reuse activity assessment. Both types of metric extraction are necessary for a complete assessment of the reuse activity in organizations.

The approach followed for evaluating the reuse potential metric focuses on trying to establish a relation between aspects such as the size of the repository and the target set of artifacts and the results yielded by the metric extraction process, as described in the next Section.

6.1.3 ADMIRE Evaluation

This Section describes in details the methodology adopted for the experiments of the ADMIRE environment evaluation, based on the considerations of the previous Sections. The goal, the configuration, the variations and the visualization of the results of the experiment are detailed.

It is important to stress that the methodology adopted, although innovative due to the nature of the proposed environment, is strongly inspired by existing methodologies for information retrieval [Baeza-Yates & Ribeiro-Neto, 1999] and reuse metrics [Basili et al., 1996] and the necessary adaptations are discussed in Sections 6.1.1 and 6.1.2.

The Goal

The goal of the experiments is basically trying to answer these questions:

• “Does the extracted reuse potential metric reflect the actual similarity between projects?”;
6.1. METHODOLOGY

- “What is the behavior of the reuse potential metric in relation to varying aspects such as the size of the index and the target set of artifacts?”; and
- “How effective is the underlying information retrieval mechanism?”.

The reuse potential metric extraction process must be evaluated in different configurations, using the artifacts of the selected projects as both the legacy code repository and the target set of artifacts under metric extraction. This strategy will be used in order to facilitate the determination of the actual similarity between two selected set of artifacts and compare it with the result yielded by the extraction process.

A reuse potential value greater than 0.2 is considered high given the absence of expansion manipulations and the presence of contraction manipulations for the underlying information retrieval mechanism, as discussed in Section 5.5.1 and the additional manipulation performed on the result set by limiting it to the most relevant result, as discussed in Section 5.5.2.

By evaluating the reuse potential extraction process, the parsing and analysis functions used for indexing and query formulation are indirectly evaluated as well. They must properly reflect the contents of a set of artifacts in order to the extraction process correctly determine the level of similarity, and thus the reuse potential, between a set of artifacts and a reuse repository. This part of the experiment can be considered as the coarse-grained evaluation of the environment.

The fine-grained evaluation will be based on traditional information retrieval evaluation [Baeza-Yates & Ribeiro-Neto, 1999], as discussed in Section 6.1.1.

To answer the aforementioned questions, the following null hypotheses are formulated:

- \( H_0a \): The reuse potential metric is not symmetric
- \( H_0b \): The reuse potential metric is not reflexive
- \( H_0c \): The reuse potential metric is not transitive
- \( H_0d \): The recall of the reuse potential metric is greater than its precision

By rejecting these hypotheses, the following alternative hypotheses are favored:

- \( H_1 \): The reuse potential metric is symmetric
- \( H_2 \): The reuse potential metric is reflexive
• $H_3$: The reuse potential metric is transitive

• $H_4$: The precision of the reuse potential metric is greater than its recall

Configuration

Metric extraction will be performed on all possible permutations with repetition of pairs of artifacts sets from the set of selected projects. For each permutation pair, one artifact set will act as the legacy code repository and the other one as the target. Furthermore, different values for the query success threshold value on the metric extraction process will be used in order to estimate the stability of the metric according to variations on the threshold value. For instance, an experiment with 2 projects (project A and project B) would have 4 different permutations: (1) project A as both the index and the target project, (2) project A as the index and project B as the target project, (3) project B as the index and project A as the target project and (4) project B as both the index and the target project. Figure 6.2 illustrates this scenario.

Figure 6.2: Possible pair permutations for projects A and B

Once the reuse potential metric is calculated for each permutation pair, a manual inspection by sampling the target artifacts, manually formulating an executing the queries and checking the result set in order to calculate the precision obtained by the information retrieval mechanism will be performed.
6.1. METHODOLOGY

The collected results will be checked in order to determine the symmetry, reflexivity and transitivity properties of the proposed metric, through its derived reuse potential relation, as defined in Section 5.3, in conjunction with the provided query formulation, query execution and result evaluation functions.

Variables

Experiments usually employ a set of variables that link causes and effects [Perry et al., 2000]. Independent variables are attributes that are actively manipulated when comparing different situations. Dependent variables, on the flip side, are the outputs whose values are expected to change according to changes to the independent variables.

In the context of this experiment, the independent variables are: project size, repository size and threshold. The dependent variables are: reuse potential, precision and recall.

The behavior of the reuse potential metric in conjunction with the provided parsing and analysis elements used in the indexing and query formulation phases will be analyzed by varying the size of the index and the target project and also varying the threshold value used by the result evaluation function.

For the size of the index and target project, two configurations will be used. The first configuration will work on a higher granularity using the modules of the projects as the artifact sets under evaluation. The second configuration will use the projects themselves as the artifact sets, thus working on a lower granularity.

For the threshold value, two configurations will also be used. The first configuration will be stricter about the similarity level between artifact sets using a threshold value of 1. The second configuration will relax the similarity criteria, using a threshold value of 0.5. Table 6.1 summarizes the variations to be performed during the experiments.

<table>
<thead>
<tr>
<th>#</th>
<th>Project Size</th>
<th>Repository Size</th>
<th>Threshold</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>module</td>
<td>module</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>module</td>
<td>module</td>
<td>0.5</td>
</tr>
<tr>
<td>3</td>
<td>project</td>
<td>project</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>project</td>
<td>project</td>
<td>0.5</td>
</tr>
</tbody>
</table>
CHAPTER 6. ADMIRE, THE EVALUATION

Results Visualization

For the visualization of the results, two different techniques will be used. The first technique, used for the evaluation of the precision and recall of the information retrieval mechanism, follows the traditional information retrieval approach, as presented in [Garcia et al., 2006], by plotting interpolated precision and recall values for varying number of retrieved documents (reuse candidates in the context of this work).

The second technique follows the approach adopted by Gall et al. [Gall et al., 1999] for structural program understanding used in maintenance support systems. This approach makes use of a color scale to represent numerical values of particular attributes (in the context of this work, the reuse potential value) with the goal of making the data comparison process a perceptive task (i.e. visual comparison) instead of a cognitive task (i.e. numerical comparison). According to Gall et al., perceptive tasks are performed more quickly, especially for large datasets. Since the environment is intended to operate on large sets of artifacts, this approach is suitable for the proposed solution.

A reuse potential matrix will be created for each configuration, where the lines correspond to the reuse potential values of all entities against a particular entity as the repository and the rows correspond to the reuse potential of a particular entity against all entities as the repository. A color temperature scheme will be used in order to make the reuse potential identification even more intuitive: cold temperature colors (from black to dark gray in grayscale) correspond to low reuse potential levels, while hot temperature colors (from light gray to white in grayscale) correspond to high reuse potential levels.

6.2 Selected Projects

The four projects used in the evaluation were randomly selected from the portfolio of Java projects of a midsized development organization, as described in Section 6.1. They cover a broad range of domains and were all written in the Java programming language.

Three of the projects are small (projects A, B and D) and one of the projects is midsized (project C). Table 6.2 summarizes the statistics of the selected projects.

The projects cover different domains and employed technologies, as described below:

- Project A is a web-based legal system for supporting the management of legal
6.3 Results

This Section presents and discusses the results of the experiments with all different configurations, as described in Section 6.1.3. A large variation on the threshold value was used in order to determine the sensitivity of the metric according to such variations. The reuse potential between all possible pairs of artifact sets was collected and the reuse potential matrices, with temperature colors corresponding to the reuse potential (similarity) between the projects, were rendered. Figure 6.3 shows the reuse potential matrix for the threshold value of 1.0 at the module level and Figure 6.4 shows the reuse potential matrix for threshold value 0.5 at the module level.
level. Figures [6.5 and 6.6] show the *reuse potential matrix* at the project level for *threshold* values of 1.0 and 0.5, respectively.

Each row on the matrices correspond to the *reuse potential* (*rp*) of all other artifact sets, acting as the *collection of artifacts* (*cart*), against a particular artifact set, working as the *reuse repository* (*rr*). For instance, the topmost row in Figure [6.5] corresponds to the *rp* of all projects against the *rr* of project *D*.

On the other hand, each column on the matrices correspond to the *rp* of an artifact set, working as the *cart*, against all other artifact sets, working as the *rr*. For instance, the rightmost column in Figure [6.4] corresponds to the *rp* of module *D7* against the *rr* of all other modules.

From an organizational perspective, such matrices are very useful as a starting point for a detailed analysis of the reuse potential among its internal projects. The cost for having such matrices is related to the time and resources necessary for indexing, formulating and executing the queries for all project combinations. An incremental approach may be employed to alleviate this overhead, as further discussed in Section [7.3].

Only approximately 11% of the methods were considered *relevant* according to the definition presented in Section [5.5.1]. This finding demonstrates that such type of *contracting manipulation* (defined in Section [3.1]) can be a powerful tool in maximizing the *precision* of the information retrieval mechanism and minimizing the size of the index as well as the volume of submitted queries to the environment by both *metric extraction* and *active information delivery* mechanisms.

This finding is also in concordance with the general rule that states that the performance of information retrieval mechanisms can be improved by taking advantage of domain-specific knowledge [Nagypal, 2005]. In the context of this work, specific Java programming language knowledge was employed to achieve this improvement.

Some important conclusions can be drawn from the collected data. The *first aspect* to be noted is that since *precision* is a priority for the solution, as stated in Section [5.1.2], the *reuse potential* between the selected artifact sets in pairs was in general very low. This requires a more detailed analysis on the contents of the sets to determine whether these are the expected results for the metric.

A further analysis on the contents of the selected projects reveals that the very low achieved *reuse potential* values are also due to the nature of the different selected projects. Some of the projects were written in Portuguese while some were written in English. Additionally, some modules perform very specific tasks, such as providing database access to the systems or interfacing with a SMS system. These modules present a very low reuse potential because of their own nature of serving
Figure 6.3: Module level reuse potential matrix for threshold = 1.0. P1 corresponds to the reuse potential (rp) between modules A2 (acting as the reuse repository - rr) and D2 (acting as the collection of artifacts - cart): between 0.1 and 0.2. P2 corresponds to the rp between modules A3 (rr) and A8 (cart): between 0.2 and 0.3. P3 corresponds to the rp of module A9 to itself (both rr and cart): between 0.5 and 0.6. P4 corresponds to the rp between modules D3 (rr) and A9 (cart): between 0.0 and 0.1.
CHAPTER 6. ADMIRE, THE EVALUATION

Figure 6.4: Module level reuse potential matrix for threshold = 0.5

Figure 6.5: Project level reuse potential matrix for threshold = 1.0
6.3. RESULTS

Figure 6.6: Project level reuse potential matrix for threshold = 0.5

very specific purposes.

The second aspect is that the metric results showed stable according to the threshold, since a 100% variation on its value resulted in small changes to the reuse potential matrix among the artifact sets.

An analysis of the resulting matrix shows that the reuse potential relation can be considered as symmetric, but neither reflexive nor transitive. Thus, from the three reuse potential metric related null hypotheses, $H_0a$ is successfully rejected, while $H_0b$ and $H_0c$ fail to be rejected. Consequently, only the alternative hypothesis $H_1$ is favored. This happens because the behavior of the analyzer is slightly different during indexing phase and searching phase, as detailed in Section 5.4.4.

The failure of the proposed metric in conjunction with the provided core functions to hold the reflexivity and transitivity properties prevents important optimizations during the metric extraction process, especially when building reuse potential matrices. Were those properties held, far less extraction steps would be necessary, since the remaining steps could be inferred from the extracted set. For instance, only half of the extraction steps would be necessary just because of the reflexivity property, since the reuse potential of a module against another module could be extracted in only one direction and the opposite direction could be directly derived from its result.
Such optimizations constitute important mechanisms to reduce the cost of creating and maintaining reuse potential matrices, as discussed earlier in this Section.

The precision and recall of the underlying information retrieval mechanism were calculated from the results of 20 queries randomly selected by the contents of the known projects existent on the repository index. Table 6.3 presents the calculated values for these metrics. Since the mean recall is not greater than the mean precision (even considering the standard deviation and variance), it can be concluded that the null hypothesis $H_0$ is successfully rejected, favoring the alternate hypothesis $H_4$.

Table 6.3: Precision and recall of the information retrieval mechanism

<table>
<thead>
<tr>
<th>Metric</th>
<th>Mean</th>
<th>Std. Dev.</th>
<th>Variance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Precision</td>
<td>0.8976</td>
<td>0.2515</td>
<td>0.0632</td>
</tr>
<tr>
<td>Recall</td>
<td>0.3055</td>
<td>0.2821</td>
<td>0.0796</td>
</tr>
</tbody>
</table>

Hence, the precision of the information retrieval mechanism is greater than its recall. This is the expected behavior, since it is one of the requirements of the solution as stated in Section 5.1.2. The achieved mean precision value is high when compared to results reported by [Garcia et al., 2006] because the queries were directly derived from the code of the known projects, which are also part of the repository index.

This approach was used to overcome the author’s lack of expertise on the selected projects, as described in Section 6.1.1. Real usage scenarios, where the working project is not part of the index, will probably yield less precise results since the main source of precision loss is polysemy [Jensen, 2004] (words with multiple meanings) and different projects can use the same words with different meanings.

The mean recall value was low because no expansion manipulation was used. According to the analyzed data, two expansion manipulations that would increase recall without sacrificing precision in the context of the selected projects would be synonym and translation handling. Part of the projects were in Portuguese and the other part were in English. Additionally, different words were used for the same concept among projects.

### 6.4 Summary

This Chapter presented the experiments conducted to evaluate the ADMIRE environment in terms of the precision and recall of its information retrieval mechanism and the behavior of the proposed reuse potential metric on a set of real projects. The results showed that although some tuning for the information retrieval engine
is still needed, the environment consists on a valuable tool for aiding development organizations in achieving a higher reuse activity level.

Next Chapter concludes this dissertation by summarizing the analysis performed on this Chapter, reviewing some related works, pointing directions for future enhancements to the environment and presenting some final considerations regarding this project.
Chapter 7

Conclusion

Once the proposed environment has been defined, implemented and evaluated based on the findings of the research on the software reuse, reuse metrics and information retrieval areas, some conclusions and comparisons can be drawn and directions to future work pointed out.

This chapter is organized as follows: Section 7.1 summarizes the findings of the work and Section 7.2 presents a comparison with some related works. Section 7.3 points some directions for future enhancements and research opportunities unexplored by this work and, finally, Section 7.4 contains a concluding discussion on the topics covered in this dissertation.

7.1 Summary of the Findings

This Section summarizes the findings of this work. These findings are enumerated below:

- Existing reuse programs are still far too remote from current practice of development organizations. This poses a great overhead for the adoption of such processes (Section 2.5);

- Software reuse and information retrieval are synergetic fields. Reuse metrics can build on this synergy to provide an accurate estimate of the reuse activity (Section 1);

- Active information delivery consists on an important mechanism for closing the gap of the reusable software location problem (Section 3.4);
7.2. RELATED WORKS

- The ADMIRE environment is an extensible framework that was built from the ground up strongly based on the reuse of existing technologies, with the goal of closing the gaps identified during the research phase (Chapter 5); and

- The evaluation results show that although some fine tuning is still needed, the ADMIRE environment is likely to be a promising tool for an enhanced reuse activity (Chapter 6).

7.2  Related Works

The products of this work are a result of a careful research on three interrelated fields: software reuse, information retrieval and software reuse metrics. The ADMIRE environment builds on these three fields to provide an integrated set of functionalities from both individual (developer) and organizational (manager) perspectives aiming at promoting the reuse activity in organizations.

Although no existing solution is integrally equivalent to the proposed solution, this Section discusses some related works on these three fields that have in some way inspired the definition of the ADMIRE environment.

7.2.1  Software Reuse Processes

Traditional approaches to a systematic reuse program focus on the redefinition of development activities to contemplate reuse specific tasks. Reuse process are divided into two main categories: Domain Engineering and Software Product Lines (Section 2.4). The existing solutions on these categories fail to provide a smooth transition path from a chaotic to a systematic reuse environment.

The reuse process defined in the RiSE project [Almeida et al., 2004], the base for this work, aims at closing this gap. The set of tools and mechanisms defined intend to provide a comprehensive environment for all reuse activity improvement stages. Since it is still a work in progress, some of the tools and mechanisms being defined may have some overlapping and future adjustment will be necessary for a cohesive solution.

7.2.2  Information Retrieval

Because of the synergy of these two fields, information retrieval has been the focus of many software reuse related works [Ye, 2001] [Jensen, 2004] [Holmes, 2004]
CHAPTER 7. CONCLUSION

This work is based on a more recent research area that has focused on using the information on the artifacts under development as the source for the queries to be formulated and actively delivering the matches from the repository.

The CodeBroker system [Ye, 2001] combines the information available in Javadoc comments of Java source files with a signature matching mechanism to filter the reuse candidates from a repository of high quality Javadoc framework documentation. The system has a front-end that integrates with the Emacs text editor to actively present the matching candidates.

Discourse models are employed to provide users with mechanism for shaping the system behavior according their past experience and knowledge level. The ADMIRE system has a similar approach, but its design contemplates integration with a broader range of tools. Moreover, different types of artifacts are contemplated and the internal structure of the artifacts is used for filtering during the indexing and searching phases.

The Strathcona system [Holmes, 2004] uses the structure of Java source files as templates in the formulation of queries to a repository of framework usage examples. Although being able to automatically formulate the queries, this system does not provide an active information delivery mechanism, requiring user intervention to execute the queries. The focus of this work is mainly on the glass-box reuse type, defined in Section 2.2. The ADMIRE environment uses the structure of the working set of artifacts, Java source files in this context, in a more limited fashion. The actual matching is based on keywords and the structure of the code is only used for filtering the contents to be indexed and the contents to be added to the queries under formulation.

7.2.3 Reuse Metrics

Although many reuse metrics have been defined, few metrics extraction tools have been provided. Poulin [Poulin, 1997] has provided a basic and an advanced web based ReuCalc reuse metrics calculators for the assessment of economics oriented reuse metrics, as described in Section 4.2. For being web based solutions, no integration with development tools or environments is provided. The metrics extraction requires the user initiative and direct intervention. The reuse metric extraction tool provided by the ADMIRE environment assesses the reuse potential metric, proposed in Section 5.3, and is integrated with existing continuous integration tools in order to facilitate the metric extraction process.

The rl sofware tool [Frakes & Terry, 1996] extracts software structure oriented reuse metrics from C code. It aims at assessing the achieved internal and external reuse activity of C programs. As with the ReuCalc calculator, the rl tool requires the
user initiative and intervention in order to extract the reuse metrics. The ADMIRE metrics extraction component is designed to extract the reuse potential of any type of asset, coded in any format and written in any language.

7.3 Future Work

The initial version of the ADMIRE environment does not cover some important areas, since the goal was to demonstrate the viability of this type of environment as a solution for the low reuse activity problem as well as provide a solid basis for future extensions and improvements. Some important aspects that were left out of this initial version are enumerated:

- **Distribution of the index**: the scalability requirement will only be met when the index is distributed among servers. The Lucene search engine, used in the initial implementation of the environment, already provides this functionality. Small adjustments must be performed to take advantage of the underlying infrastructure;

- **Access Control**: access to resources available in the reuse repository may require some control policy. Although the architecture contemplates this scenario, the initial implementation provides a very simple policy that does not filter the access to such resources. More sophisticated policies must be employed in the future;

- **Traceability and Ranking**: the provided Tracer component weights all assets equally. Future implementations must integrate with a reuse activity monitor in order to give more used assets a higher priority;

- **Sophisticated Analysis Elements**: the structure and relationships of the artifacts must be considered during the analysis phase for a more accurate artifact matching. This will allow for the introduction of some *expansion manipulations* on other searching aspects such as dealing with synonyms and foreign languages, as described in Section 3.1;

- **Artifact Types Support**: new artifact types and formats must be supported in order to provide the environment with a broader coverage range;

- **Multiple Information Retrieval Mechanisms**: since no information retrieval mechanism is suited for all contexts, multiple information retrieval strategies must coexist in the environment and policy for determining which strategies to use and the priority of each one given the context and user feedback must be provided;
• **Enhanced Results Visualization**: both active information deliver and metric extraction mechanisms provide a very simple visualization of the yielded results. More sophisticated visualization techniques, such as results clustering [Osinski et al., 2004] [Jensen, 2004], must be integrated to the environment;

• **Feedback Mechanism**: the environment must make use of user actions to improve the retrieval performance and provide users with additional feedback hooks for the creation of discourse models [Ye, 2001];

• **Extraction of Additional Reuse Metrics**: a complete assessment of the reuse activity depends on the availability of different types of reuse metrics. Reuse potential metrics must be confronted with amount of reuse and economics oriented reuse metrics for this assessment;

• **Reuse Metrics History**: the level of reuse activity in an organization is assessed by different reuse metrics. By keeping a history of these metrics, deviations from the expected results can more easily be identified and tackled. The environment must provide a framework for automatically keeping the extracted metrics.

• **Reuse environments comparison**: some experiments comparing the performance of the proposed ADMIRE environment with existing solutions [Ye, 2001] [Jensen, 2004] [Holmes, 2004];

• **Additional access points**: build on the extensibility of the environment to provide additional mechanisms, such as a web-based interface, for accessing the reuse repository;

• **Query reformulation**: a queries refinement mechanism [Ye, 2001] on the passive search interfaces to be provided; and

• **Sophisticated matching mechanisms**: provide matching implementations based on association rules to retrieve similar assets to the ones being searched [Henninger, 1997] and on a better representation of the user context as presented in [Ye, 2001];

### 7.4 Concluding Remarks

This dissertation has demonstrated that although software reuse is a promising tool for an efficient software development practice, the software community is still unable to fully extract all of its benefits.
An integrated reuse environment - the *ADMIRE environment* - has been defined and implemented having extensibility as its main design principle. Integration with existing tools is a key factor in achieving the expected increment on the reuse activity.

The environment is part of a broader context, the *RiSE project*, which aims at incrementally guiding organizations from an initial chaotic reuse activity to a more systematic, higher reuse activity levels. The evaluations demonstrate that the *ADMIRE environment* can be used as a basis for providing momentum at the initial stages of this process as well acting as a supporting tool for the later stages.
Appendix A

Asset Representation

The *internal format* that represents the hierarchical structure of the assets is defined as follows:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>prefix</td>
</tr>
<tr>
<td>suffix</td>
<td>references</td>
</tr>
<tr>
<td>contents</td>
<td>explanation</td>
</tr>
<tr>
<td>metadata</td>
<td>type</td>
</tr>
<tr>
<td>owner</td>
<td>programmingLanguage</td>
</tr>
<tr>
<td>language</td>
<td>platform</td>
</tr>
<tr>
<td>parent</td>
<td>firstChild</td>
</tr>
<tr>
<td>nextSibling</td>
<td>position</td>
</tr>
<tr>
<td>offset</td>
<td></td>
</tr>
</tbody>
</table>

The attributes *name*, *prefix* and *suffix* are reserved for identifying the asset. The dependencies of the asset are represented by the *references* attribute while the *contents* attribute contains a representation of the actual contents of the asset and *explanation* represents the rationale for the asset.

The *metadata* attribute is reserved for general purpose meta information about the asset while the *type* attribute represents the level of the assets in the hierarchy. The *owner* attribute is used for access control. The *programmingLanguage* attribute is an important discriminator when retrieving reuse candidates for a given context while the *language* attribute represents the main language of the contents of the asset. The *platform* attribute is used to enrich the context while retrieving reuse candidates.

The *parent*, *firstChild* and *nextSibling* attributes are used to represent the
hierarchy and allow the navigation through the hierarchy.

The *position* and *offset* attributes are reserved for the exact positioning of the assets inside their parents.
Appendix B

Stop Words

The set of stop words used by the initial implementation of the ADMIRE environment is listed below:

<table>
<thead>
<tr>
<th>English</th>
<th>Portuguese</th>
</tr>
</thead>
<tbody>
<tr>
<td>get</td>
<td>set</td>
</tr>
<tr>
<td>add</td>
<td>remove</td>
</tr>
<tr>
<td>test</td>
<td>action</td>
</tr>
<tr>
<td>form</td>
<td>reset</td>
</tr>
<tr>
<td>excluir</td>
<td>incluir</td>
</tr>
<tr>
<td>exception</td>
<td>excecao</td>
</tr>
</tbody>
</table>

These words were selected during an initial analysis of the word frequencies in the selected projects. The most frequent words were included in the stop words set for having a small discriminating power. The set includes words in both English and Portuguese languages, since some of the projects were written in English and other projects were written in Portuguese.

Since precision is of fundamental importance to the solution, the least frequent words were not included in the stop words set, even if they also have a small discriminating power. The rationale is that such words favor precision at the expense of recall.
Bibliography


