

Semantic Component Retrieval in Software Engineering

Inauguraldissertation
zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften der
Universität Mannheim

vorgelegt von
Diplom-Informatiker Oliver Hummel
aus Neustadt an der Weinstraße

Mannheim, 2008

Dekan: Prof. Dr. Matthias Krause, Universität Mannheim
Referent: Prof. Dr. Colin Atkinson, Universität Mannheim
Korreferent: Prof. Dr. Ivica Crnkovic, Mälardalen University

Tag der mündlichen Prüfung: 11.03.2008

Abstract

In the early days of programming the concept of subroutines, and through this software reuse, was invented to spare limited hardware resources. Since then software systems have become increasingly complex and developing them would not have been possible without reusable software elements such as standard libraries and frameworks. Furthermore, other approaches commonly subsumed under the umbrella of software reuse such as product lines and design patterns have become very successful in recent years. However, there are still no software component markets available that would make buying software components as simple as buying parts in a do-it-yourself hardware store and millions of software fragments are still lying un(re)used in configuration management repositories all over the world. The literature primarily blames this on the immense effort required so far to set up and maintain searchable component repositories and the weak mechanisms available for retrieving components from them, resulting in a severe usability problem. In order to address these issues within this thesis, we developed a proactive component reuse recommendation system, naturally integrated into test-first development approaches, which is able to propose semantically appropriate, reusable components according to the specification a developer is just working on. We have implemented an appropriate system as a plugin for the well-known Eclipse IDE and demonstrated its usefulness by carrying out a case study from a popular agile development book. Furthermore, we present a precision analysis for our approach and examples of how components can be retrieved based on a simplified semantics description in terms of standard test cases.

Zusammenfassung

Zu Zeiten der ersten Programmiersprachen wurde die Idee von Unterprogrammen und damit die Idee der Wiederverwendung von Software zur Einsparung knapper Hardware-Ressourcen erdacht. Seit dieser Zeit wurden Software-Systeme immer komplexer und ihre Entwicklung wäre ohne weitere wiederverwendbare Software-Elemente wie Bibliotheken und Frameworks schlichtweg nicht mehr handhabbar. Weitere, üblicherweise unter dem Begriff Software Reuse zusammengefasste Ansätze, wie z.B. Produktlinien und Entwurfsmuster waren in den letzten Jahren ebenfalls sehr erfolgreich, gleichzeitig existieren allerdings noch immer keine Marktplätze, die das Kaufen von Software-Komponenten so einfach machen würden, wie den Einkauf von Kleinteilen in einem Heimwerkermarkt. Daher schlummern derzeit Millionen von nicht (wieder) genutzten Software-Fragmenten in Konfigurations-Management-Systemen auf der ganzen Welt. Die Fachliteratur lastet dies primär dem hohen Aufwand, der bisher für Aufbau und Instandhaltung von durchsuchbaren Komponenten-Repositories getrieben werden musste, an. Zusammen mit den ungenauen Algorithmen, wie sie bisher zum Durchsuchen solcher Komponentenspeicher zur Verfügung stehen, macht diese Tatsache die Benutzung dieser Systeme zu kompliziert und damit unattraktiv. Um diese Hürde künftig abzumildern, entwickelten wir in der vorliegenden Arbeit ein proaktives Komponenten-Empfehlungssystem, das eng an testgetriebene Entwicklungsprozesse angelehnt ist und darauf aufbauend wiederverwendbare Komponenten vorschlagen kann, die genau die Funktionalität erbringen, die ein Entwickler gerade benötigt. Wir haben das System als Plugin für die bekannte Eclipse IDE entwickelt und seine Nutzbarkeit unter Beweis gestellt, in dem wir ein Beispiel aus einem bekannten Buch über agile Entwicklung damit nachimplementiert haben. Weiterhin enthält diese Arbeit eine Analyse der Precision unseres Ansatzes sowie zahlreiche Beispiele, wie gewöhnliche Testfälle als vereinfachte semantische Beschreibung einer Komponente und als Ausgangspunkt für die Suche nach wiederverwendbaren Komponenten genutzt werden können.

Thank you...

... to my parents Bärbel and Robert Hummel for life (*and so much more*)...

... to Konrad Zuse for the computer...

... to Peter Luffy for teaching me my first program...

... to Stephan Baumann where it all got started...

... to Colin Atkinson for his faith and support...

... to my colleagues for discussions and distraction...

... to my students for their commitment...

... to Ivica Crnkovic for coming from Sweden to report on this...

... and to myself for all the rest.¹

¹ Actually, I don't know whom to thank for the universe. So, if you should find out, please let me know.

1 INTRODUCTION.....	11
1.1 Motivation.....	11
1.2 Research Objective.....	12
1.2.1 Out of scope.....	15
1.3 Research Strategy.....	15
1.4 Outline.....	17
2 FOUNDATIONS.....	19
2.1 Software Engineering Basics.....	19
2.2 Software Development Processes.....	20
2.2.1 Traditional Process Models.....	21
2.2.2 Today's Best Practice Processes.....	24
2.2.3 Agile Development.....	25
2.3 Software Verification and Validation.....	27
2.3.1 Software Testing.....	27
2.4 Software Components.....	29
2.4.1 Component-Based Development.....	31
2.4.2 Component Technologies and Service-Oriented Architectures.....	33
2.4.3 Semantic Web (Services).....	36
2.5 Software Reuse.....	38
2.5.1 The Reuse Landscape.....	39
2.5.2 Success and Failure Factors for Reuse.....	41
2.5.3 Reuse Metrics.....	43
2.6 Component-based Reuse.....	45
3 COMPONENT RETRIEVAL SO FAR.....	49
3.1 Software Component Repositories.....	50
3.1.1 Component Representation Methods.....	50
3.1.2 The Repository Problem.....	51
3.1.3 Usability.....	52

3.2 Component Retrieval Techniques.....	53
3.2.1 Information Retrieval.....	53
3.2.2 Foundations of Search Engines.....	55
3.2.3 Component Retrieval Approaches.....	56
3.2.4 Information Retrieval Methods.....	58
3.2.5 Descriptive Methods.....	59
3.2.6 Denotational Semantics Methods.....	60
3.2.7 Operational Methods.....	61
3.2.8 Structural Methods.....	62
3.2.9 Topological Methods or Ranking Approaches.....	62
3.2.10 Discussion of Classification.....	63
3.2.11 Retrieval Techniques in Use Today.....	65
3.3 Semantics in Reuse Approaches.....	66
4 THE INTERNET AS A REUSE REPOSITORY.....	69
4.1 Estimated Potential.....	70
4.1.1 Specialized Search Engines on the Web.....	73
4.2 Precise Retrieval with General-style Search Engines.....	75
4.2.1 (Meta-)searching With Google Codesearch.....	76
4.2.2 Limitations.....	77
4.3 The Build-Up of Merobase.com.....	78
4.3.1 Crawling and Index Structure.....	78
4.4 The Content of Merobase.....	80
4.5 Sharing Components over the Web.....	83
5 SEMANTIC COMPONENT SEARCHING.....	85
5.1 Use Cases for Component Search Engines.....	86
5.1.1 Speculative and Open Source Searches.....	87
5.1.2 Definitive Searches.....	89
5.1.3 Java Library Searches.....	91
5.1.4 Further Optimization Options.....	92
5.1.5 Design Recommendations based on Search Results.....	93
5.2 Specification-Based Retrieval With Extreme Harvesting.....	94
5.2.1 Process Overview.....	95
5.2.2 Component Searching – a Hybrid Approach.....	96
5.3 Component Evaluation.....	97
5.3.1 Linguistic Conformance.....	98
5.3.2 Signature Matches in Java.....	99
5.4 Result Adaptation.....	101
5.4.1 GoF Adapters.....	101
5.4.2 Limitations of the GoF Adapters.....	102
5.4.3 Parameter Permutator.....	104
5.5 Dependency Resolution.....	107
5.6 Class Ensembles.....	108
5.7 Implementation.....	110
5.7.1 Eclipse Plugin.....	110
5.7.2 Server-Side Implementation.....	113

6	PROCESS INTEGRATION.....	115
6.1	Reuse in Test-Driven Processes.....	116
6.1.1	An Extreme Programming Example.....	117
6.1.2	Extreme Reuse.....	119
6.2	Component-Driven Design with Kobra.....	121
6.2.1	Supporting Software Design With Interface Recommendations.....	124
6.3	General Design Guidelines for Successful Reuse.....	124
7	EVALUATION.....	127
7.1	Evaluation Approaches So Far.....	127
7.2	Proof of Concept.....	129
7.3	Semantic Retrieval.....	131
7.4	Precision Analysis.....	133
7.4.1	Evaluating Open Source Searches.....	134
7.4.2	Comparison of Retrieval Techniques.....	135
7.5	Case Study.....	142
8	RELATED WORK.....	147
8.1	Component Search on the Internet.....	147
8.1.1	Agora.....	148
8.1.2	Web-Based Component & Code Search.....	149
8.1.3	Web 2.0 Technologies.....	151
8.2	Other Reuse Tools.....	152
8.2.1	CodeBroker.....	152
8.2.2	RASCAL.....	152
8.2.3	CodeGenie.....	153
8.2.4	And More.....	153
8.3	Result Ranking.....	154
9	EPILOGUE.....	159
9.1	Summary.....	159
9.1.1	Contributions.....	160
9.2	Future Work.....	161
9.3	Concluding Vision.....	162
10	REFERENCES.....	165
	LIST OF FIGURES.....	181
	LIST OF TABLES.....	183
	APPENDIX A: TEST CASES.....	185

1 INTRODUCTION

*In the beginning the Universe was created.
This has made a lot of people very angry and has been widely regarded as a bad move.*
-- Douglas Adams

1.1 MOTIVATION

Ever since Konrad Zuse developed the first computer programming language called Plankalkül [Gil97] in the 1940s software systems have been growing increasingly complex. In the early days of programming when computers used to fill rooms memory was highly expensive and hence programmers invented the concept of subroutines to conserve memory. This allowed a piece of code to be called from different locations without the need to store multiple copies of identical code. Software reuse was thus invented to better manage limited hardware resources [Cle95]. But simultaneously, software developers suffered from ever increasing pressure for shorter development cycles and higher software complexity. The software itself, however, has been suffering from a whole range of problems such as bad quality or overruns of budget and time schedules. This situation became common currency during the 1960s when the so-called „software crisis“ (see e.g. [Dij72]) was recognized. Suddenly, a solution focussed on software was required. It was presented at the famous NATO conference in Garmisch in 1968 where – amongst other ideas – the term software engineering was coined and the need for engineering-like development of software was highlighted. It was Douglas McIlroy [McI68] who introduced a related vision that was inspired from other engineering disciplines: the (re-)use of pre-fabricated software parts in order to promote flourishing component marketplaces. Today, as a matter of fact, component-based software reuse is considered one of the hallmarks that would bring software engineering closer to the standard of a fully-fledged engineering discipline [Mil99]. However, as we will detail in this thesis, component-based software reuse and the required component markets still have not made their expected breakthrough.

Arguably, software and software development have become very important for our daily lives and our economy, the annual turnover of the software industry has long become a multi-billion Euro business in Germany alone: A world without computers, microprocessors and thus a world without software is not conceivable anymore. However, the years since the turn of the millennium have also confronted the software industry with some unpleasant problems such as the burst of the dot-com bubble and increasing amounts of open source software that have become available for free over the Internet. However, some analysts have already seen a possible source of the revenues for software companies in that area and some

researchers – as well as recently some companies – have recognized the potential of the Internet as the world's largest ever repository for reusable software. Given the annual turnover of the software industry even a small quantum jump (in the physical sense of the word) in reuse technologies, making the enormous amounts of code lying on the Internet or even in version control repositories of companies reusable, could save the software industry millions of Euros per year.

Unfortunately, the Internet itself is at first an amorphous mass of bits and bytes and the challenge that remains (not only for reuse technologies) is to find and utilize the information that really counts for the users (i.e. in our context the reusable components for the developers) in the large amount of data distributed over the Internet. In other words, a component search engine must understand the semantics or the meaning of components to satisfy the needs of developers well enough to make this valuable knowledge accessible. It seems odd at first glance that this is still regarded as a problem given that component-based reuse during the 1990s was as hot a topic as web search engines have become recently. One should assume that the foundations developed within these two areas should be sufficient to deal with the millions of software assets available today. However, this was obviously not the case when the research for this thesis was started since no means to search for a specific component in a version control system were available, not to mention a way to finally find a component offering specific functionality over the Internet. Even on the World-Wide Web, which contains thousands of online shops and thus should be the first choice for so-called component markets, components have so far been hard to find. Only very recently have a number of source-code search engines emerged that try to improve this situation. However, neither the research results of the 1990s nor this code search engines of the first generation are able to provide a search capability that deserves the label semantic component retrieval as we will define it below.

1.2 RESEARCH OBJECTIVE

In principle, almost all assets produced during a software development process, like for instance domain knowledge, requirements, design and source code, have the potential of being reused and accordingly, reuse has become an umbrella concept for many different techniques that all aim at the target of *“creating software systems from existing software rather than building software systems from scratch”* as defined by [Kru92]. Scholars such as Dijkstra and Parnas first realized certain aspects of McIlroy's original vision with concepts such as structured programming [Dij70] and information hiding [Par72]. The development of object-orientation [Dah66] later integrated these ideas into the current generation of programming languages. However, software reuse in the original sense that a developer can buy a component that matches the requirements of his design was still a long way from realization when this thesis was started in 2004. The following paragraphs present more details on why we believe this topic has been (and still is) worth studying although other researchers have been working on it for almost four decades. They already produced a comprehensive range of component retrieval techniques during the 1980s and 1990s (see e.g. [Mil98] & [Luc04]), but the lack of practical applications of these techniques for publicly usable (and useful) component and service repositories is compelling evidence for the lack of theoretical knowledge in this area and hence a good rationale to investigate it more closely.

Generally, strong evidence for the effectiveness of various reuse approaches have been presented in a number of publications (e.g. [Len87], [Gri93] & [Iso92]) and reuse established itself as an umbrella term for various concepts that range from reusing small snippets of code via components in the “traditional sense” to architecture-centric reuse where domain knowledge is reused in software product lines [Cle02]. In fact the latter area has received the main interest of the reuse community in recent years, while the “classical” component-based reuse seems to have fallen out of fashion after the mid 1990s. Although there was a lot of research into setting up software repositories and retrieving assets from them during the early 1980s until the mid 1990s, no functioning solution was developed that is still in use today. This is interesting, since well-known experts in this field such as Poulin claimed that the “*reuse library problem*” has been solved [Pou99b], but also argued that reuse repositories would become unmanageable by humans once they exceed a critical threshold (of about 250 components). However, Poulin obviously based these claims only on his experience and intuition, and although he was perhaps right at the time, today there are already libraries that contain thousands of assets (like the Java standard library) or repositories with hundreds of thousands of assets (like the version management repositories of large software companies) or even millions of assets (like the repository of the popular open-source hosting site Sourceforge.net). While ever growing hardware resources and improved version control systems have made it possible to store this large amount of software, no accompanying theory has been able to provide practitioners with guidelines on how to set-up or maintain such repositories for the purpose of searching and reusing their contents, although they are certainly required for such a mass of data [Fra05]. Consequently, the repository problem [Sea99], the representation problem [Fra94] and the retrieval problem [Mil98] identified in the reuse literature about ten years ago were still wide open, when the research for this dissertation was started.

Practically, all scientific work dealing with software component retrieval to date has suffered from the problem of setting up a component collection larger than a few dozens or hundreds of components. Until recently, there simply was not a larger number of components available to researchers since industry typically showed little interest in sharing their assets with external scientists. Consequently, the results published in older publications deal with repository sizes of around one hundred components and could merely demonstrate that the underlying concepts might work (take for example [Fra94] & [Pod93]) in practice. But there is no experience nor knowledge about how to effectively reuse the material stored in repositories that are some orders of magnitude larger. This is motivation enough to wonder how the recent influences of, for example, the massive amount of open-source code [Ray97] publicly available on the Internet could promote software reuse research as well as practice. Furthermore, a whole range of other technologies that could have a bearing on for software component retrieval have improved in recent years, such as for instance, higher bandwidth Internet connections, increasing processing power, the Unified Modelling Language (UML) and the quality of integrated development environments. All of these changes have taken place after the first wave of reuse approaches was published up to the late 1990s and hence the integration of these new developments has the potential to significantly improve the practice of reuse provided that “semantic” component retrieval techniques can be made available to developers.

Thus, the first objective of this thesis is to provide a thorough investigation of the state of the art in component-based software reuse and where it might have potential for improvement by taking new and

enhanced technologies (like those mentioned above) into account. Furthermore, we want to provide the foundations for software engineers to be able to exploit the large amount of software components lying in numerous repositories all over the world and turn it into a very large resource for component- (or service-) based software reuse. This goal can be broken down into a number of high-level sub-goals which are nicely subsumed by the following quote taken from [McC97]:

“Although the idea of software reuse is simple and obvious, its implementation is not. The practice of software reuse often requires a change in the corporate culture, software process, software tool set and software skill set; as well as, of course, something to reuse.”

Three key research areas in which new developments are required to improve the state of the art and to provide a holistic approach that better supports component-based software reuse and component markets can be identified in the quote above:

1. Something to reuse
2. Software tool set
3. Software process

In other words, this thesis aims to improve the theory and state-of-the-art concerning the discovery and utilization of large amounts of software components (solving the repository problem), the creation of tools to support efficient storage (solving the representation problem) and the creation of techniques and algorithms to facilitate their effective retrieval (solving the retrieval problem). To make these developments directly useful to developers, a further goal is to integrate them tightly into well-known development processes. In short, the goal is to automate as many of the steps in the so-called reuse success chain as proposed by [Fra96] as possible.

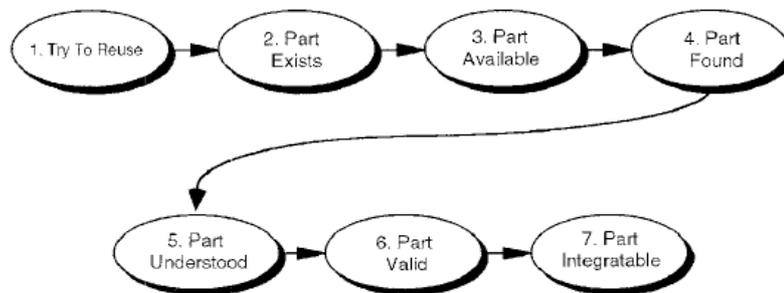


Figure 1.1: The reuse success chain [Fra96].

A special emphasis of our effort, however, is put – as suggested by the title of this thesis – on the retrieval of components by developing techniques that deserve the label *semantic component retrieval*. Since the term *semantics* is defined as “*the study of meanings*” by Merriam-Webster’s dictionary, we interpret semantic component retrieval as meaning the delivery of results (i.e. components) that have the meaning (i.e. fulfil the purpose) intended by the submitter of the query. However, the main problem in this context is the so-called *conceptual gap* (as e.g. discussed by Larman [Lar05]) lying between a concept in a developer’s mind and the actual representation of that concept in a software system or a component in a repository as discussed by [Fis91]. Thus an important contribution of the thesis is to shrink this gap and

to develop easy to use, context-sensitive component search algorithms that deliver results coming as closely as possible to meeting a developer's real desires.

1.2.1 OUT OF SCOPE

However, a well organized and searchable component repository is not only useful for component-based reuse in the traditional sense. Other applications such as improving software testing or software cost estimation and detecting plagiarism are alternative possible uses of this work and some of them we already discussed elsewhere [Hum06c]. However, since reuse is already a very complex topic in its own right, our main focus will be component-based software reuse and we will not elaborate on the areas just mentioned within the scope of this thesis. We will merely give some pointers to our further work when appropriate.

Furthermore, one would be able to identify another research area from Carma McClure's quote, namely the change in corporate culture, as a fourth point, but this is out of scope for this thesis as our focus is clearly on technical and not on managerial concerns and industrial practices as discussed e.g. in [Gri94]. Additionally, since reuse has become an umbrella term for a large collection of many different ideas, approaches and concepts it is also important to clarify that everything outside the technical improvement of software development through efficient component repositories is also out of scope for this thesis. This includes the above mentioned domain engineering and product line approaches that have been successful in practice [Cle02], as well as design patterns [GoF95], which are generally regarded as another successful attempt to provide software developers with guidelines to reuse design experience.

There are even further interesting areas in the field of software reuse such as generative programming [Cza00], software factories [Gre03] or the reuse of general software development knowledge [Bas88] that will not be considered in our work. We will only mention foundations from related areas if they are necessary for the understanding of this dissertation and refer the reader to the relevant literature for further details in those cases.

1.3 RESEARCH STRATEGY

Computer science has its roots in various disciplines like mathematics and electrical engineering and the natural sciences. Software engineering as a profession which deals with the construction of software is generally regarded as an engineering discipline [Som06]. On the other hand, it also has a scientific facet represented by software engineering research. While the mathematical approach of constructing and proving models is only of limited use for theoretical branches of computer science the growing influence of natural sciences has made empiricism very popular in software engineering [Bas86] in recent years. This paradigm is generally based on a positivist research approach which considers principles and methods of natural sciences as applicable for sciences largely influenced by human behaviour. Positivists usually start by identifying an existing problem in an area under research based on practical observations. Literature studies provide the basis to identify interesting variables and to construct a theoretical framework within which experiments can be performed. Such experiments are used to underpin or to reject the hypotheses that can be formulated with the help of the defined framework. Experiments are statistically evaluated and the outcome is used to answer the research questions and to derive principles

or laws. According to [Bla82] such a research process “*is the application of scientific method to the complex task of discovering answers (solutions) to questions*” which can be summarized as follows for social and natural – i.e. for empirically grounded – sciences:

1. choosing the research problem(s)
2. stating hypotheses
3. formulating the research design
4. gathering data
5. analysing data
6. interpreting the results so as to test hypotheses

However, although empirical research in software engineering has made a lot of progress in recent years it is not always regarded as fully sufficient. Many experiments in software engineering research involve humans and hence introduce variables in the experimental setting that are not fully controllable. For instance, natural scientists typically are able to control variables such as temperature or humidity but it is not possible for computer scientists to change the experience or age of their subjects so that methods from the social sciences have to be used to mitigate negative effects induced by humans on empirical experiments in software engineering.

Another limitation of empirical software engineering research has been raised by the advocates of the so-called design science. They argue that a pure positivist approach is not sufficient since computer science still remains an engineering discipline and finding creative solutions for identified problems should be regarded as a research paradigm on its own right since they often have to pave the way for follow up behavioural science. In other words, as described by Nunamaker and Chen [Nun90] many important developments in the history of research have been made out of creativity or necessity without any empirical research. The authors mention structured programming as well as analysis and design as examples from the area of software engineering. Another example is the development of CASE (computer aided software engineering) tools. In these areas, the development came first and the empirical studies proving their effectiveness followed some years later. Consequently, the authors realize that problems exist where empirical or mathematical solutions are not sufficient to show the success of a research approach. Since software reuse is an area that has not made any significant progress in the last decade and above all the construction of a usable CASE tool is one of the main goals of this dissertation we decided to follow the research process recommended by Nunamaker and Chen. It can be summarized as follows:

1. Construct a conceptual framework
2. Develop a system architecture
3. Analyse and design the system
4. Build the system
5. Observe and evaluate the system

The following quote from Nunamaker's article supports our decision:

“An ideal research problem is one that is new, creative, and important in the field. When the proposed solution of the research problem cannot be proven mathematically and tested empirically, or if it

proposes a new way of doing things, researchers have to develop a system to demonstrate the validity of the solution, based on the suggested new methods, techniques, or design. Once the system has been built, researchers can study its performance and the phenomena related to its use to gain insights into the research problem.“

Our research problem is the effective retrieval of software components to support software development. Due to the previous lack of usable component collections it was neither possible to test the performance of such systems empirically, nor to prove anything mathematically. We propose a combination of existing techniques to make existing component collections usable for component retrieval and hence had to develop a system that is able to demonstrate the usefulness of this concept. Only after this was defined were we able to observe and evaluate our system. This process is also reflected in the structure of this thesis which is described in more detail in the next subsection.

1.4 OUTLINE

The first chapter of this thesis has already explained the necessity for component-based software reuse approaches and identified some severe problems that could not have been solved in the past. In the preceding subsection, we explained what is in the scope of our work and, furthermore, we defined the research approach we want to follow. This choice already widely determined the further structure of the thesis. The succeeding chapter starts with some foundations on software engineering and software development processes since their understanding is required to appreciate the context of the work. After that we explain the current state of component-based software development and software reuse. We provide some more detail on common reuse approaches and their relations with each other before we discuss important aspects such as success factors for reuse, reuse metrics and the foundations of component-based development at the end of the second chapter.

Chapter three illuminates the state of the art of component-based reuse with a special focus on software component repositories and retrieval techniques. Finally, it discusses the idea of semantic component retrieval and outlines a number of conceivable use cases for semantic component searches. Since a lack of components has always stopped researchers from working with large repositories in the past, we have turned our attention towards using the Internet as a component repository. Due to the wide availability of open source software and web search engines we found it to be a feasible source for this purpose. This is described in chapter four. We explain how it is possible to use common web search engines such as Google or Yahoo for well targeted component searches although the prevailing opinion at this time was that this would not be possible at all. However, since the big search engines do not open up their indices for unlimited automated access this approach is only usable to conduct research, but not for practical usage in a production environment. Thus we discuss the creation of our own index of software components and web services in subsequent parts of this chapter.

Chapter five turns its attention to semantic retrieval techniques for software components. We first define our understanding of semantic component retrieval as delivering results with the behaviour the developer expects in a given context and we derive a number of use cases for component search engines from that statement. We then discuss how we implemented search algorithms for these use cases. In section 5.2 we

elaborate our vision of a specification-based component retrieval approach called Extreme Harvesting which is based on a syntactic description of a required component enriched with an approximate semantic description in the form of a test case. In the sequel to chapter five we discuss our implementation of Extreme Harvesting and other contributions to the state of the art that were necessary to solve the problems that occurred during the implementation of this concept. After that we demonstrate in chapter six that Extreme Harvesting fits smoothly into test-driven development processes due to its roots in the test-driven paradigm of Extreme Programming [Bec99]. We also discuss how it could satisfy the specification-based reuse approach defined in the KobrA method [Atk02] with some minor adaptations. A description of the tool we developed and the general applicability of our approach round off this chapter.

Chapter seven concentrates on the evaluation of our ideas and starts with a brief survey of attempts to evaluate previous reuse approaches. We then demonstrate the feasibility of Extreme Harvesting with an early proof of concept implementation before we present the results of a precision analysis for various retrieval techniques and Extreme Harvesting. Afterwards, in chapter eight we present the most important related work that has been carried out by other researchers in the period of time during which we were working on this dissertation. Finally, chapter nine summarizes the work we performed and highlights our most important contributions. We finish it with suggestions for future work and a vision for interactive component markets that could materialize within the next few years based on the results of our work. Finally, chapter ten contains the lists of references, figures and tables.

2 FOUNDATIONS

The nice thing about standards is that there are so many to choose from.

-- Andrew S. Tannenbaum

As this thesis aims to leverage component-based reuse for modern software development, before we can introduce the foundations of software reuse we must first define its relationship to today's common practices of software development and explain some relevant concepts and terms. Readers familiar with the basic principles of software engineering can skip the first part of this chapter and continue reading from section 2.4 where we introduce and discuss the idea of software components. Afterwards we provide an introduction on software reuse which starts in section 2.5 on page 38.

2.1 SOFTWARE ENGINEERING BASICS

Software engineering is typically concerned with the development of a system, i.e. the *product*. As defined by Endres and Rombach [End03], "*a product is a system, consisting of hardware, software, or both, to be used by people other than the developers*". A product is the result of a *project* which "*is an organizational effort over a limited period of time, staffed by people and equipped with other resources required to produce a certain result*". The chances of successfully completing a project are raised by following a *process*. The literature, e.g. [Som06], defines a software process as "*a structured set of activities required for the development of a system*". Furthermore, a software *process model* is an abstract representation of a process. It presents a simple description of a process from some particular perspective. A development *method* is more comprehensive than a process and includes a description of the development activities to be performed as well as a description of the assets to be developed.

Another important ingredient of today's software development approaches, as in other engineering disciplines, is an abstract (graphical) representation of complex designs. It took researchers and practitioners almost thirty years from the "invention" of software engineering at the famous NATO conference in Garmisch in 1968 until the late 1990s to establish the Unified Modelling Language (UML, as of 2006 available in version 2.0 [OMG04]) as a commonly accepted graphical notation for software systems. Moreover, to quote the Kobra book [Atk02], "*the Unified Modelling Language (UML)*

is currently the leading notation for modelling architecture and design level information in a graphical form". Since there is no rival on the horizon, the UML will probably remain the leading graphical software notation in the foreseeable future.

Basically, the UML offers elements for describing various aspects of a software system such as class diagrams describing the structural view of a system and interaction diagrams showing how objects communicate with each other. It is supported by the Object Constraint Language (OCL, see [War03]) which provides a semi-formal language for enhancing the precision of models and describing the semantics of a system's functionality. UML also contains component-diagrams that can be used to describe the required and provided interfaces of components (these will be described in more detail later). Since the UML is the quasi standard for software blueprints today we will use UML diagrams where appropriate to better illustrate our ideas or to depict examples. We require a basic understanding of UML from the reader. Easy to read introductions to the UML and the OCL can be found in [Fow03] and [War03] respectively.

2.2 SOFTWARE DEVELOPMENT PROCESSES

This section briefly describes common ways of developing software today. It highlights development processes that either have been of some importance for the field as a whole in the last four decades or are of high relevance for this thesis. Although the older sequential software development models are now generally recognized as being insufficient in practice, teaching in software engineering usually starts with simple software development approaches such as the waterfall model. We want to keep this tradition and also introduce the models in chronological order.

However, before we can actually go into the description of the models we have to define some further terms. Although the waterfall model still appears as a model in most modern software engineering literature it is not a *software process model* in the sense that it gives concrete guidelines on what to do when in a development process (as defined in the last section). Consequently, [Sca02] characterizes it as a *life-cycle model*, i.e. a model that depicts only a coarse-grained scheme that could be used for management purposes. In contrast to a life-cycle model, a process model has a much higher descriptiveness and can be used as a technical recipe for building software. Nowadays, developers can choose from a variety of established development processes. There is a good reason for this as there are a lot of factors that influence the success of a software project (such as developer experience in a domain, clarity of requirements, safety requirements, size of the system etc.) and only one kind of development process would never be optimal for every type of problem. There is therefore a wide range of different approaches, which can be arranged into three broad groups – namely into sequential, iterative and agile development approaches.

Independent from the chosen development process, a software's life-cycle typically comprises five groups of activities that have to be carried out to get from the initial idea to a working system:

- ◆ analysis
- ◆ design
- ◆ coding
- ◆ testing and integration
- ◆ operation and maintenance

Since most modern development approaches allow a to carry out some of these activities in a different order or concurrently, we deliberately omitted a numbering of them in the bullet list above. The following subsections will explain the arrangement of these activities in common development processes and will give some concrete examples on how they might be applied in practice. However, it is beyond the scope of this thesis to explain all these activities in detail. A lot of good books are available on each these activities such as the ones from Coad and Yourdon for object-oriented analysis [Coa90] and object-oriented design [Coa91], from Eckel for programming in Java [Eck06], from Beizer on testing [Bei90] or for maintenance [Lie80]. Textbooks [Som06] that try to cover all aspects of modern software engineering are also available as well as good books (e.g. [Lar05]) on practical, individual processes that explain how to apply these activities together.

2.2.1 TRADITIONAL PROCESS MODELS

This subsection gives a brief introduction into “classic” software development models such as the waterfall model.

SEQUENTIAL MODELS

The waterfall model (see e.g. [Boe76], [Som06]) is the classic representation of a software system's life-cycle which attempts to discretize the activities of software development. It is printed in nearly every book on software development and comprises a structured set of activities that proceed in a sequential order, each of which must be completed before the next one can start. Its first appearance [Roy70] dates back to 1970, and Royce had already suggested to apply the model iteratively at that time. However, this is no longer widely known and has given the model a rather bad name. Anyway, the waterfall model (as we said above some regard it only as a “life-cycle” model) is widely recognized as being poor in terms of descriptiveness and process guidance and is often considered as being too rigid for real life usage e.g. due to changing requirements. The figure below shows the typical representation of the model as attributed to Royce:

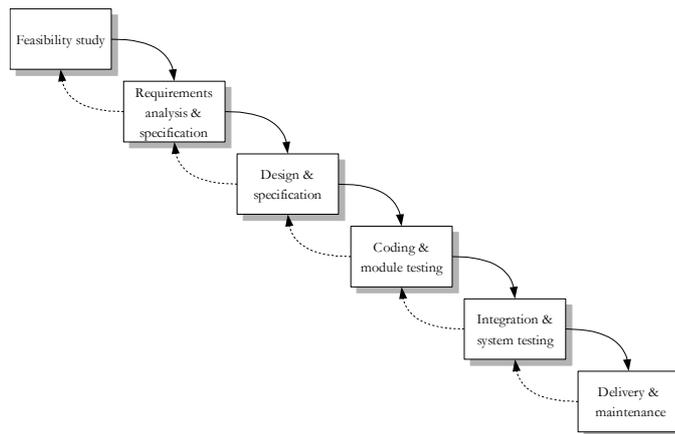


Figure 2.1: Classical waterfall model [Roy70].

The assets shown in the figure are produced in a strictly sequential manner, i.e. before system design and specification can start all requirements must have been written. Only small feedback cycles (indicated by the dotted arrows) are allowed e.g. for the correction of errors in earlier stages. For the actual creation of systems with a waterfall-style approach the initial system specification is developed through stepwise refinement into the final system. But as hinted above, there are no concrete guidelines on what activities should be used to develop the system. Furthermore, the waterfall model requires a kind of unnatural system development process since the requirements of a system tend to be unclear at the beginning of the process or tend to change after they have been written down.

The V model [Boe84] traces back to an idea of Boehm and obtained its name from the typical "V form" (see figure 2.2) that is used to characterize the ordering of the activities in the process. The V model is of particular importance in Germany as it is the compulsory model when software is to be developed for the federal administration. In principle, it is merely a waterfall model with an elaborated view of the verification and validation activities.

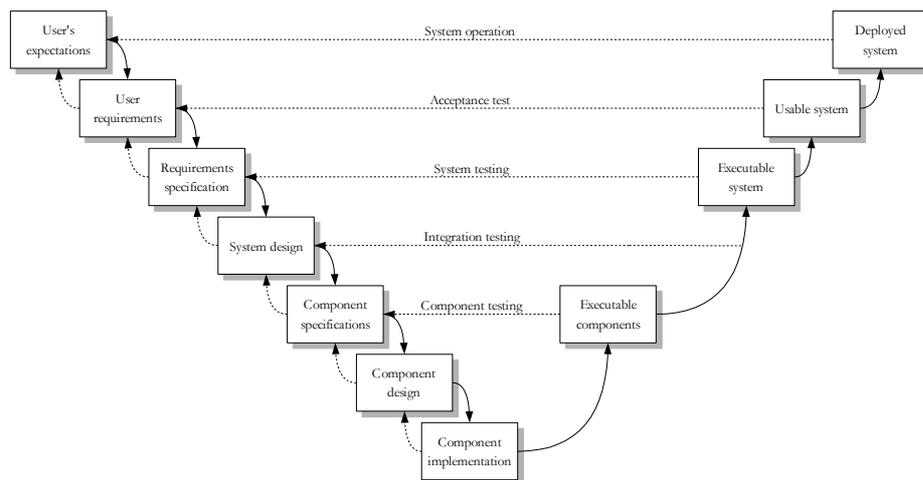


Figure 2.2: V Model of Software Engineering.

Validation (“Are we building the right system?” [Boe84]) in this model takes place in the horizontal direction, i.e. the products on the right are tested against the documents or requirements on the left. Verification (“Are we building the system right?” [Boe84]), however, takes place vertically similarly to the waterfall model where one development product p_n is verified against the p_{n-1} , i.e. the one before. Since its first version in 1986, two updated versions of the V model have been published by the German government, namely the “V-Modell 97” [VMo97] and the “V-Modell XT” [Bro05] where XT stands for “Extreme Tailoring”. The main focus of the latter, as the name suggests, is to provide a collection of building blocks that can be combined into a process model tailored to a project's needs.

ITERATIVE MODELS

Over time a lot of proposals for iterative models have been made. Initially, even the original proposal of the waterfall model intended to progress through the waterfall, iteratively. However, Boehm's spiral model [Boe88] is regarded as the archetype of iterative process models although it is not a process model in the common understanding of the term. It is rather a kind of “meta process model” that incorporates risk observations. Other models should be used inside the spiral for the actual software development. The most notable fact about this model is that it integrates iterations for the first time and hence can be seen as the predecessor of modern iterative approaches such as the Rational Unified Process (RUP) [Kru00], explained in more detail in the next subsection. The spiral in this model is separated into four quadrants that are passed through in every spiral loop. During the first quadrant (the upper left one), the objectives, alternatives and constraints of the next development step are determined and during the second the risks arising for these objectives are evaluated and dealt with. The third quadrant typically is concerned with the actual development steps like collecting requirements, creating a software design, actually coding or testing the software. The fourth quadrant is used to plan the next loop through the cycle.

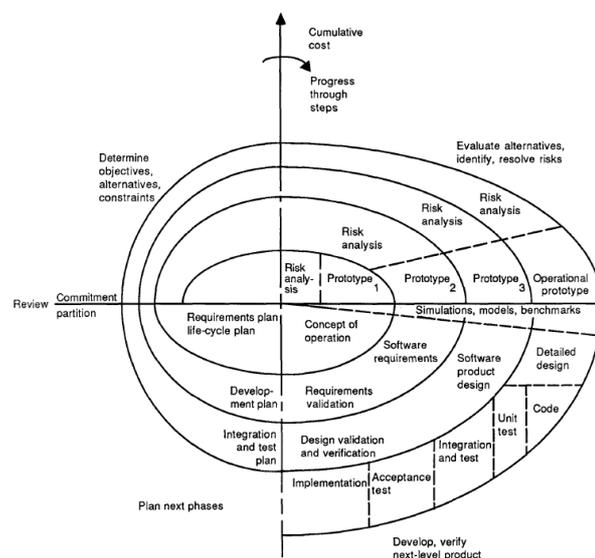


Figure 2.3: Boehm's original figure of the spiral model as in [Boe88].

2.2.2 TODAY'S BEST PRACTICE PROCESSES

The process models mentioned before are an important foundation for software development processes in use today, but are typically not ready to be used out of the box. As we later want to discuss component reuse in the context of modern software development we now give a brief overview of the development processes most used in practice today.

RATIONAL UNIFIED PROCESS

The Rational Unified Process (RUP) [Kru00] is today's de facto standard for UML-based system development in many organisations around the world. It was originally developed by Rational as a process to complement the UML. The general idea of the RUP is an iterative development approach that is illustrated in figure 2.4. At first glance, the iteration is not as obvious as in the spiral model, but the idea is still simple. The development span of a project is separated into four phases called inception, elaboration, construction and transition which follow each other sequentially as shown at the top of the figure (i.e. time flows from the left to the right in the figure). Although this might imply some similarity to the waterfall model it is important to note that this is not the case as the common workflows (or activities) listed on the left are repeatedly passed through. Each of the four phases is subdivided into iterations which are “timeboxed” and usually have a duration between 3 to 6 weeks. The coloured bulges in the central area of the figure represent the effort that is spend on a certain activity over time. For instance, at the beginning a lot of effort is put into business modelling and rough requirements elicitation – that is, most of the requirements are collected and assigned a priority and/or risk. Then they can be allocated to iterations, typically high priority/risk requirements are allocated to early iterations and hence, as the figure makes apparent, the design and implementation of important functionality can already start in the inception phase.

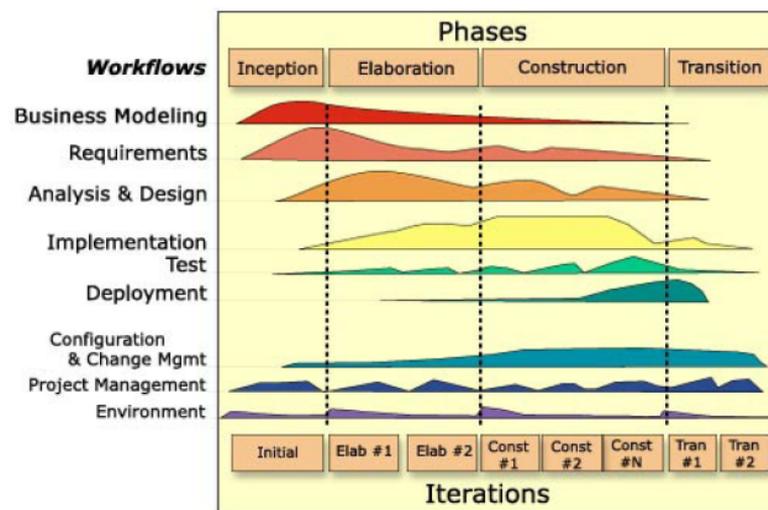


Figure 2.4: Graphical representation of the Rational Unified Process (from IBM website).

2.2.3 AGILE DEVELOPMENT

Agile approaches [Coc01] comprise a number of iterative methodologies that were first introduced as light-weight methodologies and share some important ideas with the RUP. Consequently, in practice many companies use the best elements from both and apply something like an agile RUP as is proposed by [Lar05]. In contrast to so-called heavy-weight, traditional methodologies, agile methods are supposed to be a compromise between too much process and no process at all. The mantra of agile development proponents is to have “just enough” process. Hence, agile methods are typically not very predictive in terms of cost or effort estimation, but highly adaptive and that is exactly what they are intended to be. Agile development methods are typically used for small- and medium-sized projects where it is difficult to determine all requirements at the start. The so-called agile manifesto [Fow01] is commonly accepted as the original definition of agile development and contains the following principles:

- ◆ Individuals and interactions over processes and tools
- ◆ Working software over comprehensive documentation
- ◆ Customer collaboration over contract negotiation
- ◆ Responding to change over following a plan

The manifesto was signed in 2001 by 17 prominent developers from the agile development field, including for example Kent Beck, Alistair Cockburn and Martin Fowler. Agile processes obviously work for many projects limited in size and effort, but critical observers have always pointed out that agile development is only a collection of best practices used to impose a little bit of structure on chaotic projects. Although this is certainly not said without any reason, we believe there is a rationale for the use of agile processes in certain projects with unclear and rapidly changing requirements for example.

Most agile approaches also include a recent trend in software engineering, namely the test-driven development approach. We describe this further in the following subsection because it has an important bearing on our component retrieval approach.

TEST-DRIVEN DEVELOPMENT

One of the most important hallmarks of Extreme Programming (XP, [Bec99]) and other agile methods is the so-called test-first or test-driven development approach (TDD, [Bec03]) which, to some extent, turns traditional development processes (like the waterfall model) upside down and creates test code immediately before production code rather than afterwards as was traditionally the case. The following activity diagram illustrates the typical flow of a TDD process:

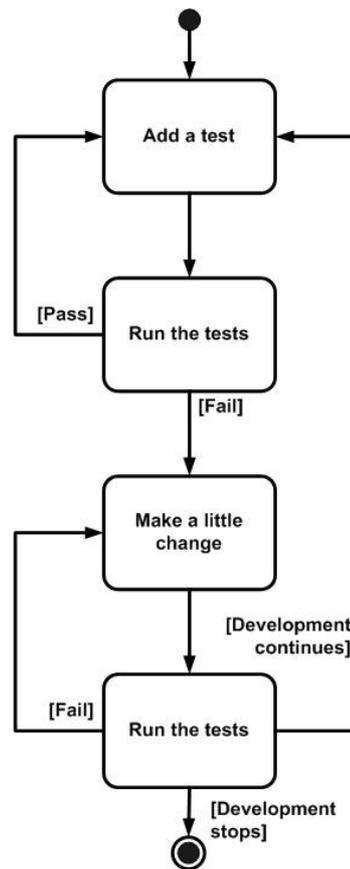


Figure 2.5: Activities in a TDD process
(source: [Amb03]).

As the name already indicates, the first step is to create a test for the unit (class) under development. Then the test is run and its failure is checked. The unit under test is then developed with the aim to make it pass the test. Therefore *“the simplest solution that might work”* [Bec03] is usually implemented. After implementing the functionality, it is tested again. If it still fails, the production code has to be reworked and retested. Alternatively, it is possible to enhance the test cases according to the principle of triangulation [Bec03] or to continue with the development of other unit tests that cover new functionality. In a nutshell, this basic development cycle is also known as *“design a little, test a little, program a little”*, a maxim that originated from Extreme Programming.

The literature (e.g. [Lar05], [Bec99], [Amb03]) lists the following key advantages of such a process incorporating TDD:

- ◆ The tests get written at all
- ◆ Programmer satisfaction
- ◆ Clarification of interface and behaviour, i.e. low level design
- ◆ Repeatable verification
- ◆ Helpful documentation

Some authors even advocate the early creation of tests after initial design work and before code is implemented in more heavyweight processes, such as Larman [Lar05] for the Unified Process. We regard this approach as very helpful since it allows – as we shall see later – a simple integration of our specification-based reuse approach.

2.3 SOFTWARE VERIFICATION AND VALIDATION

Test-driven development places the main emphasis of quality assurance on software testing. Of course, this is important in general, but software quality assurance is typically a much broader activity and not only involves product-related quality during development, but also organizational ones such as process control etc. Certification of companies according to CMMI [Chr03] is one example of the latter, but this topic is so large that we have to limit our attention to the former at this point. Software testing is normally performed as a part of software verification and validation (V&V) activities. It was Barry Boehm [Boe84] who made the famous statement that verification is about *“am I building the product right”* and validation is about *“am I building the right product”*. Software testing is a dynamic V&V technique, while e.g. software inspections are a static V&V technique. Since the notion of testing is important for this thesis, we will briefly explain its most important aspects in the following subsection. For more detailed insights on software inspections, we have to refer the reader to the literature. [Gil93] provides interesting material on inspections in general while for instance Basili et al. [Bas96] have published an interesting piece of work that is regarded as improving software inspections and also demonstrates how empirical software engineering research can be conducted effectively.

2.3.1 SOFTWARE TESTING

Software testing [Bei90] is the process of evaluating whether a software is fit for the required purpose. The general idea is to provide a program (or an operation) with a set of input and expected output values and to compare the latter with the actual values delivered by program execution. Unfortunately, as Dijkstra pointed out in [Dij72] *“program testing can be used to show the presence of bugs, but never their absence”*, testing can neither be used to prove the correctness nor the completeness of a program. Theoretically, it is possible to exhaustively test a program by comparing the result for every possible combination of input values with the expected output value. However, this so-called exhaustive testing would not only require large amounts of computation even for small pieces of software, but it also relies on the availability of a so-called oracle that is able to predict the correct output for every possible combination of input values. Obviously, if such an automated oracle existed, there would be no need to implement the system itself.

Hence, in practice a variety of techniques are used to choose representative candidates from the input space to shrink the testing effort to a bearable amount. We distinguish between two basic purposes of software testing, namely defect testing for discovering bugs in a software system and reliability testing for assuring a required level of reliability. We limit ourselves to defect testing at this point as the underlying statistical models for reliability estimation are too complex to be discussed here. Defect testing has traditionally been used at different stages during software development. Although not necessarily bound to it, the V model presented in section 2.2.1 gives a good overview of the various levels of testing that are

typically performed. More detailed information can also be found in [Som06] and the more specialized testing literature named above. So-called unit testing has the smallest granularity and typically provides test cases on the method level. When classes and components are assembled to larger functional units integration testing is performed to guarantee the proper interplay of the different parts. System testing is performed to validate the whole system before it is delivered to the customer and acceptance testing is finally performed by the customer to check whether the system fulfils his/her requirements. Testing on the unit level will be required later in this thesis so we briefly explain the two feasible approaches in the following subsections.

BLACK-BOX TESTING

Black-box (or functional) testing [Bei95] is perhaps the more intuitive testing technique. It is practised for instance in electrical engineering as well. A component is regarded as a black-box, i.e. only through its interface (or its connectors in electrical engineering). It therefore assumes that a specification of the component is available, but no details about its internal implementation. The component is tested by sending input values into the black-box and observing the output values that are returned. Those can be compared with the values expected, according to the specification. Since the internal structure is not available, and cannot be evaluated or used to derive test cases other strategies must be used to validate a black-box component. Black-box testing typically utilizes equivalence partitioning and boundary value analysis. The former is used to reduce the number of test cases to a manageable amount some it is obviously not possible to test for instance the 2^{64} possible input values of a simple operation such as `add(int, int):int`. Hence a tester tries to cover at least each equivalence partition (like positive and negative numbers and zero for this example) with some meaningful samples. This is where boundary value analysis comes into play as it is commonly accepted that boundary values at the edges of equivalence partitions are places where faults most frequently occur. Obviously, black-box testing makes most sense with binary components or services where access to the source code is not possible. The more flexible approach of white-box testing (see next paragraph) can only be used when the source code is available.

WHITE-BOX TESTING

White-box testing (also known as structural testing, [Mye02]) needs access to the internal structure of components, i.e. the source code has to be available. Using this information, test cases can be tailored to fulfil a given code coverage criterion such as, for example, attempting to execute every statement. This is called statement coverage, the most simple coverage criterion, which for 100% coverage requires that a test case traverses each statement in the code at least once. However, it is somewhat fragile as the following simple Java example demonstrates:

```
String s = null;
if (some condition)
    s = "SWT";
System.out.println(s.length());
```

A test case that would carry out the example and set the condition to true would reach a statement coverage of 100%, but if the condition would once be evaluated to false in the real program, the `println` statement would cause a null pointer exception. The so-called branch (or condition) coverage avoids this pitfall as it requires that every condition is evaluated to true and to false. But even branch coverage does not guarantee that all possible errors will be disclosed and thus various other extensions (such as Multiple Condition Coverage [Mye02]) have been proposed. Ultimately, only the so-called path coverage criterion guarantees that really all possible paths through a program have been covered. Unfortunately, every pass through a loop is regarded as a path in its own right and consequently the number of paths can quickly become large and non-testable in practice. A simpler approximation has been introduced for the practical usage of this technique that only requires to cover every independent path in a program. The number of independent paths and therewith the minimum number of test cases required to cover all independent paths in a program is determined by its cyclomatic complexity [McC76] and is calculated by adding one to the number of branches in the code. The cyclomatic complexity is also a well-known code metric indicating the complexity of a piece of code.

2.4 SOFTWARE COMPONENTS

Interestingly, all the development approaches discussed above were initially focused on development from scratch and thus none of them is concerned about concrete guidelines for reusing pre-produced software parts. Approaches for componentization of software that go beyond the usual “divide and conquer” aspects of architectural and object-oriented design are not considered. Others have recognized these weaknesses and have proposed approaches that try to address this problem. For instance, [Moh04] tried to incorporate reuse into the RUP. However, the applicability of these ideas is rather limited without appropriate tool support and even the more specialized approaches that we discuss below are not yet for practical use out of the box at the time of writing.

It was in 1972 when Parnas [Par72] first wrote about managing the complexity of software systems by decomposing them into modules to simplify maintenance and development as well as to foster reusability. Since then, programming languages have made considerable progress and object-oriented languages today seem to provide all the necessary mechanism for the componentization of software. Moreover, component technologies such as J2EE, .NET, CORBA and Web Services are now widely available and the underlying languages are based on decades of experience with object-orientation (since [Dah66]) and information hiding [Par72]. However, although quite a number of definitions for “software component” have been proposed in recent years, there is often confusion about what this term means, especially in relation to the term “object”. This might be stem from the fact that even the modern component technologies mentioned above do not fully comply with the common definitions for components presented below.

Probably the most popular definition of the term “component” can be found in [Szy02] and originates from a workshop of the 1996 European Conference on Object-Oriented Programming:

“A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.”

Certainly, this definition gives a kind of lower bound for components, and object abstractions commonly used in modern programming languages are supposed to fulfil this definition. They exhibit their public operations to the outside and can be displaced to another environment. However, none of the common programming languages support objects which fully expose their required interfaces. For example, required libraries are usually hidden in the component's implementation. But this is not an argument against treating objects as components since Enterprise Java Beans (EJB) for example are widely accepted as a typical incarnation of components although they often comprise just one object. And even Syzperski himself makes somewhat contradictory statements when he argues on the one hand without direct relation to the above definition that objects cannot be considered as components ([Szy02] on page 38) and claims on the other hand on page 285 that *“a (Java) bean is really a component”*. Since a bean is nothing but a (simple) Java class, there is obviously a problem with these statements. The Object Management Group (OMG) provides another similar definition in [OMG03] which avoids the above-mentioned problem with the required interfaces and thus also accepts objects as components:

“A component represents a modular, deployable and replaceable part of a system that encapsulates implementation and exposes a set of interfaces.”

Apparently, these two definitions are very similar, but also both raise another question: What is *not* a component? For this thesis we take the view that objects in modern programming languages comply with these definitions. Even a static method can be acceptable as a component since it could be removed from the surrounding object and placed elsewhere. Having outlined the minimum requirements for a component, one might wonder whether there is also an upper bound for this concept? Nowadays most component-based development approaches such as KobrA [Atk02] express the opinion that *“one man's system could be another man's component”* and compose components hierarchically into larger components which hide their implementations behind their interfaces as we shall see in the next subsection. Thus, they accept that there is no general upper bound for this concept. The problem in implementing this with today's object-oriented programming languages is, however, that objects are typically grouped in packages which cannot have an interface on their own. They just act as a simple container and are thus not a component in the proper sense. However, for example in Java it is possible to mimic a component's behaviour to a certain extent with the use of inner classes. Although hierarchical composition of inner classes is possible in Java it requires invasive changes of the source code and thus is no longer consistent with the intent of the definition. This issue currently makes it difficult if not impossible to find components according to a KobrA specification beyond the class level.

One other fact that should be intuitively clear in this context is the more comprehensive a component becomes, the more difficult it becomes to deploy it in a different environment without any modifications or to find a matching component in a repository in the first place. Taking these issue into account, a third definition, originating from Ralf Reussner and cited after [Kra03], becomes worth mentioning in this context:

“A software component is an artefact of the software development process and can be deployed in several contexts by third parties without being manually modified.”

In our opinion this definition is very useful since it brings in the aspect of not being manually modified. We believe that this should and will become a very important feature in supporting component markets of the future. As soon as a developer can buy a component that will automatically made fit into his system the incentive to use such a component is much higher than if he has to create adopters or glue code by hand. This definition is certainly closely related to the term “reusability”, which is defined in [McC97] as *“the extent to which a software component can be used with or without changes in multiple software systems, versions or implementations”*. In other words, if it requires no effort to reuse a component in various systems its reusability is the highest. However, according to this definition a higher reusability can also be achieved by sophisticated tool support.

2.4.1 COMPONENT-BASED DEVELOPMENT

Given the size of today's software systems a “divide and conquer” approach is an absolute necessity in order to distribute the development effort amongst numerous developers (sometimes numbering in the hundreds). Consequently, almost all modern development approaches contain guidelines and activities for architectural decomposition that maximizes cohesion and minimizes coupling of a system's parts. These parts might be called objects, components, packages or units, but in general they can all be subsumed under the common notion of “component” just introduced. Given the number of modern development approaches available today – many of which are even called component-oriented – one would assume that these processes contain guidelines on when and how to acquire components instead of developing them from scratch. However, this is not the case. Whether it be the RUP [Kru00] or KobrA [Atk02], none of these methods provides concrete guidelines on how to reuse components or where to find them.

We have chosen KobrA (abbreviation for: **K**OMponenten**B**asie**R**te **A**nwendungsentwicklung which is German for component-based application development) to explain in more detail the principle concepts in component-based development. We explain KobrA here as a state of the art development method that contains an explicit focus on component-based development and thus could smoothly host the reuse approach which is developed later in this thesis without any major modifications. We will explain the integration and application of our Extreme Harvesting approach in KobrA later in section 6.2. KobrA was initially developed at the Fraunhofer IESE in Kaiserslautern, Germany, and is comprehensively documented in the “KobrA book” [Atk02]. KobrA's primary goal is to facilitate the development of more cost effective and higher quality software systems through a component-based, reuse-oriented paradigm based on the UML [OMG04]. The developers of KobrA adapted best practice ideas from other development methodologies like Fusion [Col94] to develop a comprehensive method that fulfils four basic objectives, namely to be a simple, systematic, scalable and practical approach. A condensed and updated description of KobrA was published in the context of the Common Component Modelling Example (COCOME, [Atk07]).

The KobrA methodology is focused on the analysis and design phases of development and if a system is developed from scratch it applies a top-down development approach. Starting with an abstract

description of the whole system, it is recursively decomposed until the level of plain data objects is reached. A complete model driven description of a system is created in KobrA by hierarchically organizing the models of the individual components which it contains. The position of a component in the hierarchy is determined by the logical containment structure. Every (behaviour-rich) object in KobrA is regarded as a KobrA component (so-called “Komponent”) according to the so-called principle of uniformity. The basic idea governing the use of the UML in KobrA is that individual diagrams should focus on the description of the properties of an individual component and only those diagrams should be produced that are really needed. The former is known as the principle of locality, the latter as the principle of parsimony. Fig. 2.6 shows how a rich business component is modelled in KobrA by means of a suite of tightly related UML diagrams. We explain in section 6.2 how reusable components can be considered in a KobrA-based process. The following figure shows the various views that are to create to describe a KobrA component.

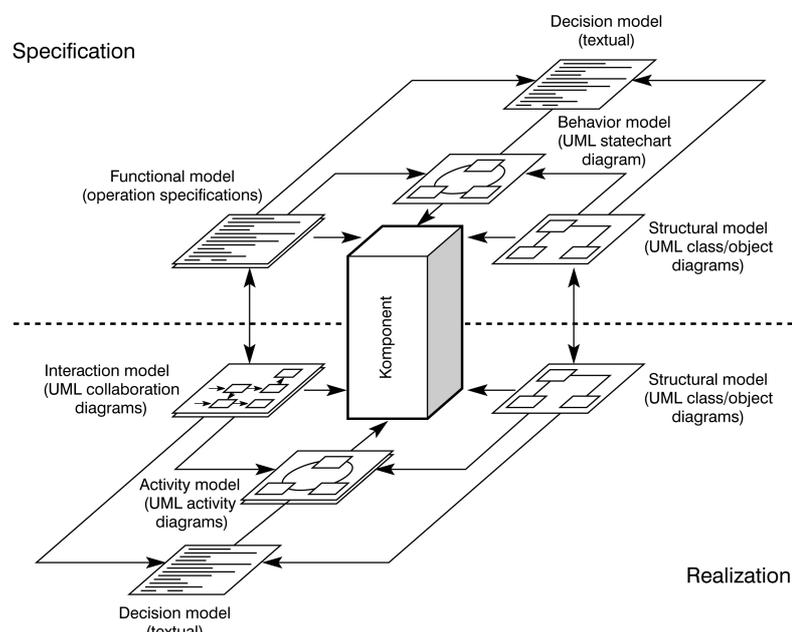


Figure 2.6: KobrA's component model [Atk02].

The specification diagrams collectively define the externally visible properties of the component, and thus in a general sense can be viewed as representing its interface. This is called the principle of encapsulation which is based on the information hiding principle proposed by Parnas [Par72]. The structural diagram (UML class diagrams) describes the types which the component manipulates, the other components with which it interacts and the list of services and attributes which it exports. The functional model provides a declarative description (i.e. contracts) of each of the services or operations supported by the component in terms of pre and post conditions. Finally, the behavioural model describes the externally visible states exhibited by the component, typically described with UML statecharts. The decision model shown in the figure is used to support various configurations for product line engineering.

The realization diagrams collectively define how a component realizes its specifications in terms of interactions with other components and objects. This can include externally acquired server components, or subcomponents which the component creates and manages itself. The realization diagrams collectively describe the architecture and/or design of the component. The structural diagram is a refinement of the specification structural diagram which includes the additional types and roles needed to realize the component. The interaction diagrams document how each operation of the component is realized in terms of interactions with other components and objects. Finally, the activity diagrams document the algorithms by which the operations are realized.

2.4.2 COMPONENT TECHNOLOGIES AND SERVICE-ORIENTED ARCHITECTURES

Today's component technologies are not only a means to package functionality they also provide support for inter-process and even inter-machine communications between components, commonly known as middleware. Basically all component technologies and models available today operate according to the client-server principle. This is similar to object technology where the object that offers a service is regarded as the server and the one that requests a service is seen as the client. According to [Wei01] a component model is the basic prerequisite for a component-oriented system and the crucial difference between components and objects is that the former conform to a component model. They require a component model to define "*standards for component implementation, naming, interoperability, customization, composition, evolution and deployment*". However, we find it difficult to follow this argument since these features should certainly also exist in a sophisticated object-oriented environment. Basically, three technology standards have been competing in this area in recent years. Both big programming platform vendors (i.e. Microsoft and Sun) provide their own standard to support componentization and communication of different processes running perhaps on physically different machines. Microsoft's approach became known as COM [Box97] and has been transferred into the .NET framework while Sun extended the idea of Java Beans and remote method invocation (RMI) in standard Java towards Enterprise Java Beans (EJB, [Sun01]) in the Java Enterprise Edition. The Object Management Group (OMG) as the third big player in object technology contributed a platform independent component technology named CORBA [OMG00]. Bindings for CORBA have become available for all major programming languages. With the recently developed web service standards a fourth player has entered the stage that can be considered a component technology as well. This also brought a new development paradigm, namely the service-oriented architecture (SOA), which is supposed to extend component-based development. Without doubt, web services have added a great simplification to distributed cross-platform computing, but in terms of componentization we regard them as a step back towards the object-oriented or perhaps even procedural development paradigms.

CORBA

CORBA, the OMG's Common Object Request Broker [OMG00], is not a language or a platform itself. This is demonstrated by the abstract so-called Interface Definition Language (IDL) which can only be used to describe interfaces for components in CORBA but does not offer the possibility to implement any functionality. CORBA's main purpose is to enable the development of distributed systems across platforms. Thus CORBA requires the use of so-called mappings that connect elements from the IDL to elements in concrete programming languages. This basic principle of operation is identical to other

distributed object technologies and web services, of course. On the client side the definition of the server's interface must be available and a so-called stub is created. The stub is a facade that forwards invocations to the underlying middleware, the Object Request Broker (ORB) in the case of CORBA, which transmits calls over the network. On the server side another ORB instance receives the requests and forwards them to the implementation of the server code. The IDL basically offers the same concepts as most other modern object-oriented programming languages, namely modules for grouping and scoping and interfaces that define the functionality available. Thus, the mapping of complex (KobrA) components to CORBA is difficult. Only recently did the CORBA Component Model (CCM, [Wan01]) present a UML profile for CORBA components that is supposed to address this weakness and some other limitations contained in the original CORBA standard. However, since the latter was developed for UML 1.5 it became outdated with the introduction of the component diagrams in UML 2.0. To our knowledge an update of the CCM to UML 2.0 was still an open issue at the time of writing.

EJB

Sun's Java 2 Enterprise Edition and the associated Enterprise Java Beans (EJBs) are available in version 3 [Sun06] and have grown into a full enterprise application framework that supports packaging and remote execution of components as well as persistence mechanisms. This, however, requires an application server since EJBs are normal Java classes and their special features can only be used inside such a container. While the EJB versions prior to 3 included a so-called XML-based deployment descriptor that defined the interfaces of an EJB, the current version of the EJB specification has moved away from that concept and tried to simplify their description. EJBs now have become "plain old Java objects" (POJOs) and their interfaces are described accordingly with plain old Java interfaces and not in an XML file anymore.

.NET

Microsoft's Component Object Model (COM) was developed during the 1990s and recently became part of the .NET framework. Its prime target platform is of course Windows, but there are implementations for other platforms as well (such as Mono open source project for Linux). COM is designed to create objects and to communicate with other processes beyond the boundaries of various Microsoft programming languages. This has been made possible by a binary format that must be shared by all supporting languages. A large range of other technologies are connected with COM and often COM is seen as an umbrella term for them. Examples include the Distributed Component Object Model (DCOM) and OLE (for Object Linking and Embedding) and ActiveX controls that all offer a way to reuse chunks of functionality. COM applications are built from COM-aware components that expose interfaces with globally unique interface IDs and versioning information. This is one of the main advantages over Sun's EJB model which does not provide a versioning mechanism or support for unique identification (only hierarchical packaging based on Internet domain names is recommended by Sun).

WEB SERVICES

With the recent advent of web services the idea of service oriented architecture (SOA) became popular. We use the term SOA as a slightly more general synonym for the term web service architecture that can also be found in the literature. The World Wide Web Consortium (W3C) defines web services as a *"software system to support interoperable machine-to-machine interaction over a network"* [W3C04]. This is

by no means a new idea. The underlying concept of remote procedure calls (RPC) is about 30 years old and nowadays integrated in almost every modern programming language (e.g. RMI on java-based platforms). Before web services became available each platform had its own proprietary way for managing RPCs and interoperability between different platforms was difficult to achieve. Even the CORBA initiative of the OMG was not able to really relieve this problem since different CORBA implementations were sometimes not able to work with each other smoothly.

On the contrary, a web services architecture reorganizes a system and its infrastructure into a set of loosely coupled, cooperating services and requires the use of three core well-defined standards, described in the following, to achieve this goal of interoperability. To quote the W3C [W3C04] another time:

“Web services can be generally defined as loosely coupled, reusable software components that semantically encapsulate discrete functionality and are distributed and programmatically accessible over standard Internet protocols.”

This definition makes the close relationship between components and services clear and thus we regard the latter simply as a descendant of the former and try to investigate both whenever possible in this dissertation. However, the long-term vision of SOAs is to automate service selection (and therewith service reuse) to the greatest extent possible, and the semantic web community [Ber01] for example has already made some valuable – although widely theoretical – progress towards this goal as we will briefly discuss in section 2.4.3. The general vision is captured by the famous UDDI triangle shown in the following figure:

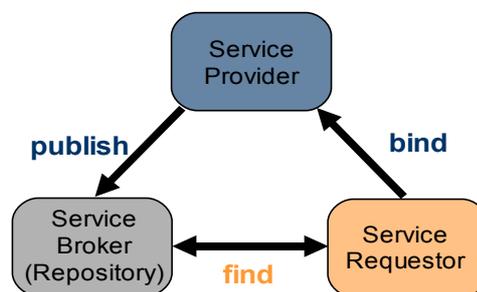


Figure 2.7: Web service brokerage architecture.

The figure illustrates the three players in a basic service-oriented architecture scenario. The service provider offers a service and registers it with the service broker where it can be discovered by the service requester who wants to use a service. It is easy to recognize the similarity to the scenario of retrieving a component from a repository. The main difference at this point is that within a SOA-based system the access to a service (i.e. a functionality) is mediated directly while a component repository only offers components that normally have to be downloaded, if necessary compiled and deployed before they can be used.

UDDI

The abstract structure in figure 2.7 illustrates the basic idea behind the architecture of UDDI, an industry initiative for the **U**niversal **D**escription, **D**iscovery and **I**ntegration of software services [New02]. The register of a business in a UDDI repository consists of three separate elements called

white, yellow and green pages. The white pages contain things such as the address and contact information for the service offered, the yellow pages contain a categorization of the service according to standard taxonomies and the green pages, which are the most interesting from a technical point of view, contain the information about how a service can be invoked over the Internet. In principle, UDDI is supposed to contain all elements necessary for successful brokerage operations allowing users to find, pay for and access a service. However, this has not been borne out in practice. A prime example for the shortcomings of the UDDI model was provided by the surprising shut-down of the so-called Universal Business Registry (UBR), which we discuss in section 4.1.1.

WSDL

The Web Service Description Language (WSDL, [New02]) is used to describe the syntactic interface of web services in an abstract way. Hence, it reveals the same information about a web service's functions as for example a Java class does about its public methods. Based on XML [New02], WSDL also provides the information necessary to communicate with a web service, i.e. the message formats expected by the service and the protocol bindings used to exchange the messages. For actually calling a service SOAP, described in the next paragraph, can be used. One drawback of WSDL is that it does not contain semantic information that would fit into the vision of the semantic web [Ber01]. However, several enhancements have been already proposed to bridge this gap, probably the most well-known of which is Web Ontology Language (OWL, [Ant04]) and its descendant OWL-S for the semantic mark-up of services.

SOAP

After discovering a service via UDDI and exploring its description with WSDL, SOAP [New02] is the means to finally access the service over standard Internet protocols like HTTP or SMTP, which is why it works far better within firewalls than the competing protocols of CORBA, RMI or DCOM. SOAP (formerly an acronym for Simple Object Access Protocol) is XML-based and used to send the messages, i.e. typically the parameters and return values of a remote procedure, defined in the WSDL file.

According to a recent survey of Hurwitz and Associates [Bar06], the main expectations driving investment into SOA are reuse and interoperability. While the latter has started to take off, the former still has not, which is not least demonstrated by the above mentioned failure of the UBR. This may sound surprising, since the UDDI is a fairly sophisticated service model and has been a part of the approach from the beginning, but as we will show in the following subsections there are more factors influencing reuse than just the availability of basic tool support.

2.4.3 SEMANTIC WEB (SERVICES)

Proposed by Sir Tim Berners-Lee et al. in their famous Scientific American article [Ber01] the semantic web is proposed as an extension that makes the current World Wide Web understandable for computers. It is closely related with numerous concepts of information retrieval and artificial intelligence and hence is briefly discussed in this section. The basic idea of the Semantic Web is to annotate web pages (and the natural language contained in them) with machine processable information. Today a variety of languages mostly based on XML and other techniques is under discussion for this purpose as shown in the well known layer cake diagram depicting the architecture of the Semantic Web below.

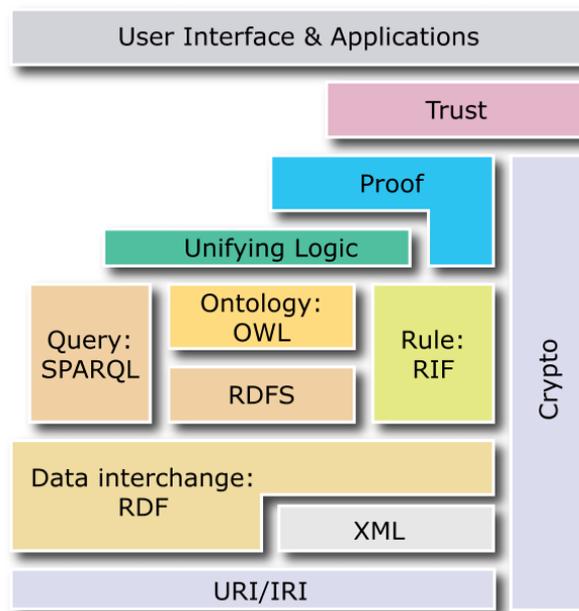


Figure 2.8: W3C's Semantic Web layer cake diagram.

Of interest for the remainder of this subsection are the Resource Description Format (RDF [Las99]) and the Web Ontology Language (OWL [Ant04]). RDF is a simple language for representing objects and their relationships in “sentences” comprising subject - predicate - object triplets. RDF Schema [Bri03] adds the vocabulary for describing properties and classes and thus the capability to describe ontologies. However, for this purpose normally OWL is used since it is more expressive and adds formal semantics.

However, explaining all these ideas in more detail would go far beyond the scope of this thesis and thus we refer the reader to the literature [Fen05] and merely briefly explain at this point how ontologies are supposed to support reuse in the context of the so-called semantic web services [Car05]. A common ontology is probably the most important factor for a successful introduction of semantic web services. The literature (e.g. the two books we mentioned above) typically describes an ontology as a static data model for knowledge representation, which contains concepts and their relationships within the world or just a specific domain. Therefore it typically contains concepts with attributes and relations between them. In the terms of software engineering it is certainly not wrong to describe an ontology as an inheritance hierarchy, although this term might not be totally sufficient. However, it cannot be denied that there is a certain similarity between the applications of OWL and UML class diagrams and one might raise the question of how far these two might be representable by each other. More details on this interesting question can be found in [Kik05].

Semantic web services typically use ontologies for composing systems out of more fine-grained services. In other words they aim to implement exactly the component reuse idea of McIlroy [McI68] for services with very little human interaction. In order to discover and match usable web services for a given purpose, typically the parameters and return values of a service are described with the help of the ontology. In theory, the main advantage is that it is possible to reason about such an ontology and thus to find not only direct matches, but matches with a different structure or those that require a

combination of some smaller services to deliver the required functionality, as well. However, we are still not aware of any practical application of these ideas and it seems that semantic web services share exactly the same problems as components, namely the repository problem (i.e. collecting enough semantically enriched services), the representation problem (i.e. find one suitable ontology (language) that is understood and used by everybody) and a combination of the retrieval and usability problem (i.e. how to formulate a query in a user friendly way). We will discuss this in more detail in chapter 3. The latter problem is a general problem of ontologies though. Either they are not complete i.e. not all necessary concepts of a domain are modelled in the ontology or they are so complex that it is almost impossible to find the right concept within the ontology. Thus, it will be interesting to see in the future if and when these ideas will materialize into usable products. It is already interesting that articles have been published recently warning that the research community is losing sight of the main goal of semantic web services, namely the automated discovery and integration [Shi07].

2.5 SOFTWARE REUSE

We already pointed out in the introduction that the idea of software reuse is as old as software engineering itself. Although it is also a rather simple idea, a lot of different definitions of this term have been proposed over the years. To pin down the common element of most reuse definitions in the literature we present Krueger's well-known definition [Kru92] at this point since most others are very similar to this:

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

This means that, in general, it is imaginable that assets from all phases of the software development process can be reused. [Fra96] provides the following table of potentially reusable artefacts from software projects:

1. architectures	6. estimates (templates)
2. source code	7. human interfaces
3. data	8. plans
4. designs	9. requirements
5. documentation	10. test cases

Table 2.1: Potentially reusable aspects of software projects according to [Fra96].

In accordance with this table from [Bas88] we see a reuse potential for all assets associated with a software project. For example we are aware of approaches for reusing software requirements [Lam98], domain knowledge [Pri91b] or even large parts of software systems in so-called product lines [Cle02]. Mili et al., however, determine in [Mil02] that reuse traditionally meant the reuse of code fragments and components. Interestingly, this was a hot topic in the research community during the 1980s and 1990s and even some success stories were highlighted at this time (e.g. [Len87]), but these systems never became practically useful and have become outdated by the size of today's standard libraries. Hence it is

no surprise that mainstream interest has turned towards more successful approaches such as product lines and design patterns [GoF95]. It seems plausible that similar to fault discovery, the earlier in the development process an asset and its successors can be reused, the more benefit can be derived from it. The practical success of the more architecture-centric reuse approaches such as the just mentioned product-line engineering certainly seems to provide initial confirmation of this claim. On the other hand, to date, it has not been investigated whether the reuse of software requirements will lead to any benefit or how such an approach could be technically implemented. Requirements on custom made software vary significantly from customer to customer and given the statements in many older publications (such as Krueger [Kru92]) that abstraction is one important prerequisite for reuse, it is questionable whether the requirements on software are a good basis for a reuse approach. Consequently, Krueger saw the lack of good abstractions at that time as one explanation for the lack of successful reuse programs. Even today, ten years after the UML was introduced, abstraction mechanisms for handy software pieces are still an active area of research.

The expected benefits of reusing software assets and knowledge [Bas91] are quite obvious and can already be found in many textbooks about software engineering, although there are few practical confirmations of this claim. Sommerville [Som06] for example, like many others, draws comparisons with other engineering disciplines and points out that mechanical and electrical engineering projects base their designs largely on reusable components that have been extensively tested in other systems. This approach looks appealing for software, too – plugging software together from prefabricated parts to produce working systems of higher quality in shorter periods of time without inventing the wheel over and over again. Moreover, since software engineering has successfully adopted ideas such as design patterns [GoF95] or separation of concerns [Kic97] from other engineering fields, component reuse should be transferable to software engineering as well. At face value, the theory sounds very appealing: There are thousands of reusable functions in software libraries, thousands of objects and components in software repositories and at least hundreds of software product lines in large companies around the world. How could there still be a problem?

2.5.1 THE REUSE LANDSCAPE

In order to classify component retrieval approaches and their place in the family of reuse approaches we briefly discuss some important concepts in the following paragraphs. These high-level descriptions are widely based on the classic textbook by Sommerville [Som06] complemented with pointers to the original publications or our own observations where appropriate.

Naturally, the classical component-based reuse approach also has its place within this classification. Sommerville identifies three different granularities of reusable software units, in order of decreasing benefit and complexity:

1. System or application reuse
2. Component reuse
3. Object and function reuse

A more detailed consideration of these concepts reveals that all three are more or less in common practice these days and many successful examples of their use can be found. Functions are in fact the smallest units that it is conceivable to reuse with today's technology, since they just about fulfil the definition of a component given by Szyperski [Szy02] (we already discussed this issue in section 2.4). And of course, function reuse has been done for many decades as evidenced by libraries like the C standard library or the Java class Math. However, other APIs offered by the Java Development Kit (JDK) for instance, go one step further. They contain all kinds of reusable packages like Java-3D etc., which do not fit the level of object reuse any more. However, they are not components in the classical sense and thus show the limitations of this classification and demand an extension, which we will introduce below. Complete applications or at least parts of them have been reused, which is typically captured under the term "commercial off the shelf" reuse discussed in the next paragraph. Our own slightly more detailed classification which has been inspired from a presentation of Morisio in 2006, comprises the following levels, also ordered according to decreasing complexity:

1. Commercial off the shelf – reuse of whole applications
2. Component-based reuse – aiming on components in the sense of Kobra
3. (Object) repository-based reuse – search and retrieval from a dedicated repository
4. Library-based reuse – browsing in class and function libraries
5. Code reuse – [Kru92] called this code scavenging

When we talk about the reuse of *commercial off the shelf* (or COTS, see e.g. [Voa98]) components, we typically mean whole (end-user) applications such as typical desktop software, database systems etc., which are normally used "as is" out of the box. It often seems difficult to include such applications in custom built applications since they are typically not adaptable and their APIs are sometimes not even documented at all so that they have to be glued into the system with scripting languages, for example. [Wei01] call this a "*lack of granularity*" which leads to attempts to factor out the more fine-grained elements (i.e. components) from these applications to increase reusability. Another term that we briefly need to mention is the term *business component* which is another term that has only vaguely defined semantics. It is generally defined as a software component that offers functionality for a business domain, see e.g. [Car01] or [Tur02] in the German business computing community. Thus they can be viewed as a specialization of the general term component as defined above. Similar to testing, we talk about *black box reuse* if only the specification of a component or a service is available and its implementation is hidden, unmodifiable behind its interface. Likewise, *white box reuse* occurs when the internals (i.e. normally the source code) of a component are available and can be altered. Furthermore, the idea of glass box reuse has been introduced by [Nea96], meaning that developers are able to behold the source or interface of a component, but only use it to learn from it and not to modify it.

2.5.2 SUCCESS AND FAILURE FACTORS FOR REUSE

Contrary to the visions of McIlroy and others, software reuse is still at a rather rudimentary level and it is not possible to buy components in the same way that you can by for example screws from a do-it-yourself store. As long as there have been publications about reuse there have been publications asking why reuse has not worked properly in practice. Some authors speculate about the reasons and justify them with personal observations [Puo99], others conducted surveys [Fra95] and finally there are some that have performed mature failure mode analyses [Mor02]. Interestingly enough, is difficult to find two publications that agree with one another on the reasons for the low level of component reuse in practice. Consider, for example, the paper by Frakes and Fox in which they asked “*sixteen questions about reuse*” [Fra95] and let us use it as the starting point for a brief review of success and failure factors. In the early 1990s the authors conducted a survey with 113 software professionals from 29 (mainly US-based) organisations. They readily admit that this is not a good random sample, but nevertheless it can give at least some hints about factors influencing reuse levels at that time. However, it is important to mention that reuse in this survey was regarded as a very generic concept and thus certainly more research is necessary to take into account today's state of the art tools such as software search engines and proactive recommendation tools, for instance. The following figure summarizes the results of Frakes and Fox and is taken from their article:

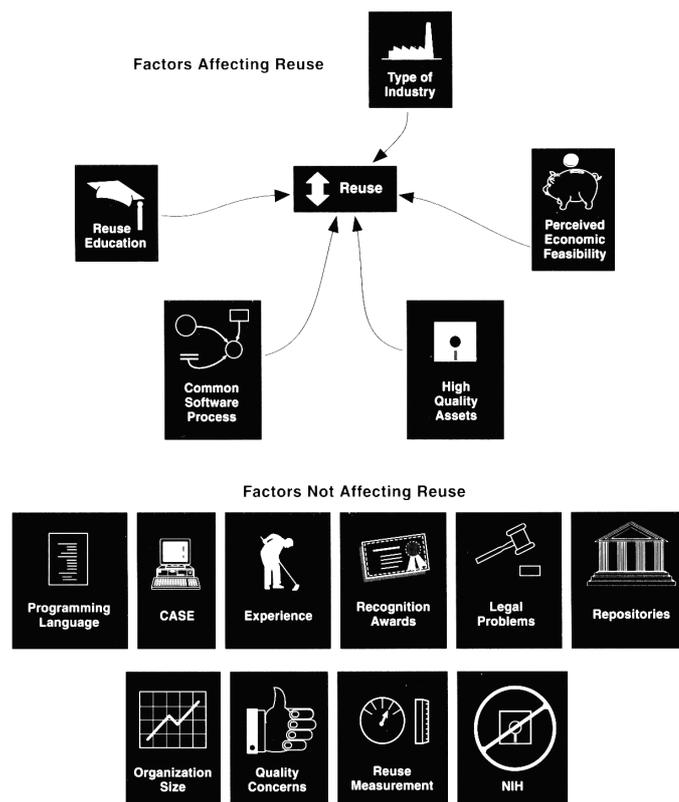


Figure 2.9: Overview of which factors affect reuse levels and which do not.

As depicted in the figure, the survey identified in only five reasons recognized as influencing reuse in a general way. Training for reuse seems to be an important factor on reuse. This is plausible since developers who are not aware of reuse possibilities will probably not reuse, and in a more general sense is acknowledged by other publications like [Mor02] and [Is092] who recommend a top management commitment and thus a reuse-friendly environment for companies. The perceived economic feasibility is also a plausible obstacle to reuse, but is perhaps overvalued in this survey since no developer would claim the contrary in a questionnaire. It is also interesting that [Fra95] identify the use of a common software process as a factor affecting reuse since even today there is practically no widely-used development process well-suited for component reuse. However, [Mor02] sees processes adapted for reuse as one decisive success factor. The type of industry is also recognized as a success factor by [Fra95]. This is clearly covered by [Mor02] who found that the type of software under production influences reuse levels considerably.

The programming language, on the other hand, does not seem to influence reuse levels a lot. This is an opinion which is backed up by our observations that all modern (object-oriented) programming languages supporting information hiding provide more or less the same levels of support for reusability. However, older research has often been carried out with functional languages (as in [Zar95]) that are free of side-effects and built upon a strict type system. This might be an additional indicator for enhanced support for reuse from this family of languages since we are only aware of one substantial work that tried to transfer this knowledge into the object-oriented domain [Str94]. We can conclude that functional languages might be better suited for reuse, but since they are not in wide-spread use today this advantage is quickly outweighed by the small number of reusable components available. In general, it is assumed that the higher the degree of abstraction supported by a programming language the better its support for reuse [Fra95]. This seems to be acknowledged by the fact that object-oriented programs are generally much more reusable than equivalent programs in procedural or even in assembly languages. However, the recent MDA approach [Bas03] still has to substantiate this claim for the even more abstract model level.

While [Fra95] does not recognize repositories as an affecting factor, [Mor02] argue that although having a repository is not sufficient for a successful reuse program, but an effective repository is nevertheless usually required. This makes sense given the demand for high quality assets that is seen as affecting reuse by [Fra95]. The latter paper also rules out CASE tools as an influencing factor, which is certainly plausible as there were no CASE tools with tight reuse integration as recommended by [Ye01] at the time of their survey (and hardly anything we would call a CASE tool from today's point of view). Both publications agree again when they exclude an organisation's size and incentives (i.e. rewards) as factors influencing reuse. The latter is covered by other findings [Fra96b] that developers are generally motivated to do a good job and choose the option (i.e. reuse or no reuse) that is perceived as the more promising. This might also explain why quality concerns about reusable assets were also ruled out. As soon as no good reusable software is available developer build their systems from scratch and vice versa. This is also closely related to the so-called not invented here (NIH) syndrome, which is often mentioned in the literature (e.g. [Gri93], [Faf94]) and normally used as a generic term for various other human factors for why developers might avoid reusing components (such as the steep learning curve needed to understand acquired components or concerns about their quality). While [Fra95] did not perceive

developer experience as important for reuse, [Des06] observe from their recent survey that “*reuse works better among novice than expert developers*”. Potential legal problems and reuse measurement are also not widely seen as influencing reuse levels. However, especially the former might change in the near future due to the large amount of reusable open source software with many forms of open source licenses and sometimes subtle legal issues to consider.

According to e.g. [Fra96b] many developers do not reuse because they do not even try. Any of the reasons discussed in other parts of this section might influence this decision, but it is apparent that a developer's motivation to reuse might decrease drastically once he has tried to reuse a few times and was not able to find suitable items, an opinion which is also backed up by the well-known and astonishingly simple observation by Prieto-Diaz [Pri87]: “*To reuse a component you first have to find it*”. [Fra96b] presented the following chain of actions that underlines this presumption. This can be viewed as a list of things that can fail during the process of reusing a component. This process is described as follows:

1. No Attempt to Reuse at All
2. Part does not exist
3. Part is not available
4. Part is not found
5. Part is not understood
6. Part is not valid
7. Part cannot be integrated

Understandably, the further a developer gets in this process the more time he probably will have invested into finding and adapting a component and the more frustrated he is likely to become if his endeavours finally fail. If this happens to a particular developer more than once or twice his motivation to try to reuse in the future is likely to be significantly reduced.

In this subsection we have presented and discussed a large number of factors that might affect reuse or that usually influence the attitude of developers towards reuse. Given the fact that most papers in this area to date are either based on rather weak empirical numbers or no empirical observations at all, there is still a lot of debate on which factors influence reuse levels and which not. In almost all cases one can find papers that hold one view or the other. The only thing that there is general agreement about is the fact that developers are likely to reuse only when they are aware of the possibilities and regard it as more cost effective than building software from scratch. From this we believe it is safe to conclude that developers will reuse software if there are tools available that effectively support them in finding and integrating reusable assets. Or in other words, components will be reused only if the effort to integrate them is smaller than the effort to develop the functionality from scratch [Pri87].

2.5.3 REUSE METRICS

The success of all new ideas and methods needed to be evaluated in practice and consequently the reuse community has been thinking of metrics to measure the degree of reuse in a system. Frakes [Fra96] defines a metric as “*a quantitative indicator of an attribute of a thing*”. According to this publication, a *model* should also capture the relationship between particular metrics. Probably the most important metric for a software asset in this context is its *reusability* if it is built from scratch (or **for** reuse) and the

amount of reuse if it is built **with** reuse [Sam97]. While the latter is relatively simple to calculate by the number of reused lines of code divided by the total lines of code (as long as we do not care about (re-) used elements from frameworks etc.), reusability is much harder to assess. For white box reuse this obviously seems to be related to source code complexity metrics such as Halstead's program volume [Hal77] or McCabe's cyclomatic complexity [McC77] as suggested e.g. by [Cal91] who developed a basic reusability model, but to date it is not clear what the relation is. The reusability of black box components is probably mainly influenced by their interface and documentation, but again at the time of writing there is a clear lack of understanding of how the interface of a reusable component would look compared to a not so reusable component, for instance.

Figure 2.10 provides a schematic overview of the two metrics mentioned above and four other conventional metrics related to reuse as proposed by [Fra96]:

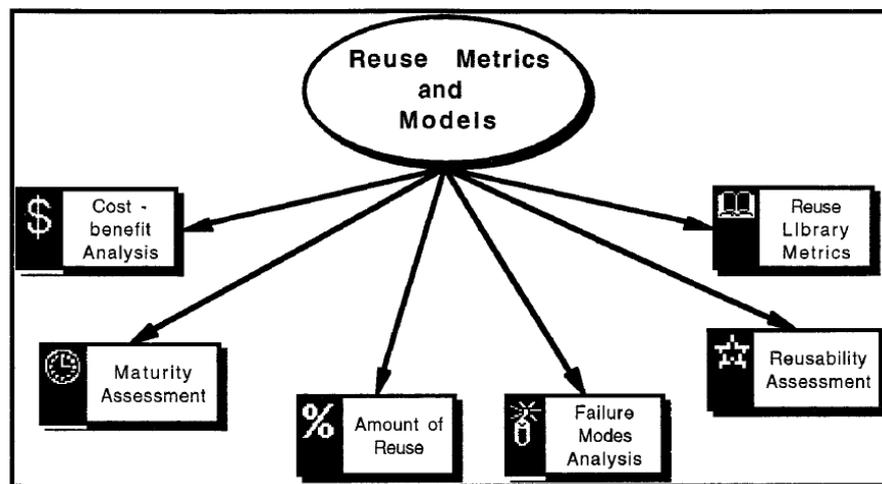


Figure 2.10: Categorization of reuse metrics and according models from [Fra96].

Cost-benefit models cover the economic aspects of reuse including benefit analysis and productivity pay-off. The maturity assessment metrics are about estimating the maturity of a reuse approach. Failure mode analysis is commonly used to find elements that impede reuse in an organization and to compare their severity. As already mentioned above, reusability assessment is used to estimate the likelihood that a given artefact is reusable. Finally reuse library metrics cover the data that accrues when a reuse repository is used.

At the end of the day, as mentioned above, reuse has to pay off, i.e. developing with reusable assets has to be more effective than developing from scratch [Pri87]. “More effective” usually means cheaper, faster, better (i.e. with less errors) or two or even all three of them. Gaffney and Durek [Gaf89] proposed a simple economic model to make this tangible through the following equation:

$$C = 1(1 - R) + Rb \quad \text{or} \quad C = 1 + R(b - 1)$$

where C is the relative cost of developing a software product (i.e. $C = 1$ for a software component developed completely from scratch). R denotes the proportion of reused code and b is the relative cost (e.g. for searching and adapting) of the reused code portion. If C is smaller than 1 reuse is considered to

be more cost effective than developing from scratch. This model is obviously also applicable to the development time and the number of errors in a system and can be extended to

$$C = (1 - R)1 + (b + \frac{E}{n})R \quad \text{or} \quad C = (b + \frac{E}{n} - 1)R + 1$$

when we incorporate the development effort E (typically > 1 since creating a reusable component is expected to be more expensive) for a reusable component. n denotes the number of uses over which the cost of the reusable component are amortized. [Mil02] have collected a number of results from the literature that indicate the relative cost of developing an asset for reuse as being between 1.10 and 2.0 times as high as for the same asset not optimized for reusability.

Interestingly, at the time of writing there are still no such numbers for developing a system with reusable components. This might be another hint that so far no sufficiently usable reuse systems have been developed. Thus, as with most models in software engineering, the central problem with the reuse metrics introduced in this section is still the difficulty of applying them in practice as predictive models. Due to a lack of empirical data and sometimes even due to a lack of understanding and clear definitions no values are so far available to use in these formulas.

2.6 COMPONENT-BASED REUSE

The idea of component-based reuse has been around for almost four decades and thus it might be surprising that the literature contains very little information about how to apply it to software development in practice. Today, however, virtually every software development project contains reuse in one form or the other. Every high-level programming language is shipped with standard libraries that offer important functionality for use out of the box. Mature frameworks exist for many purposes and developers take them for granted in their everyday work. Interestingly, most standard libraries and frameworks have become larger than even the most sophisticated software repositories developed even up to 10 years ago. Even generic data types such as a List that were examples for sophisticated reuse about ten years ago, can be parametrized to hold integers, floats or any other required type and are contained in programming languages such as Java by default today. However, this was still of interest to the reuse community about one and a half decade ago [Bas91]. Even “real reuse” occurs frequently in many projects when developers use general web search engines or the new code search engines to find source code snippets that could help them solve a problem. This so-called code scavenging [Kru92], however, is discouraged by the anti-pattern book [Bro98], for example, since it is supposed to degrade the design of a system in the long run. This begs the question whether there is a distinction between “good reuse” and “bad reuse” and how the former should be incorporated into a development process without compromising the final product?

Sommerville provides a first abstract summary of what a reuse-oriented and component-based development process could look like in his well-known textbook. This is shown in the following figure:

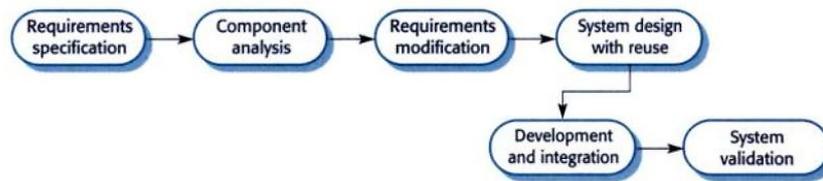


Figure 2.11: A simple component-based software development process as proposed by [Som06].

This approach is based on classic sequential process models such as the V model. While the initial requirements specification is not impacted by the component-based reuse approach the next step is. After the requirements have been specified, in a reuse-oriented process the set of available components is searched and the potentially most useful candidates are selected. Due to the complexity of software components there will typically be no complete match and hence it makes sense to consider slight adaptations of the requirements in the next step. As changed requirements can in turn have effects on the candidate components the component search and analysis phase may need to be performed again. Once an acceptable combination of requirements and available components has been found a system design is created that includes the interfaces of the candidate components as well as any necessary glue code and parts that have to be developed from scratch. Once the design is finished missing parts of the system can be developed and integrated with the available components. Finally, system validation can be performed in the usual way.

In addition to this general description, dedicated component-based development methods such as KobrA [Atk02] offer a more idealistic approach, which takes effect later in the development process and is based on the availability of a large repository of candidate components. KobrA proposes component search and retrieval based on the specification of a component. As described earlier, KobrA specifies a component as a black box and merely describes its externally visible features such as a syntactic and semantic interface description. Based on the initial description KobrA includes an iterative negotiation process for adapting the originally required interface with the one actually offered by the candidate component if no direct match is achievable. The general idea is shown in the following sketch:

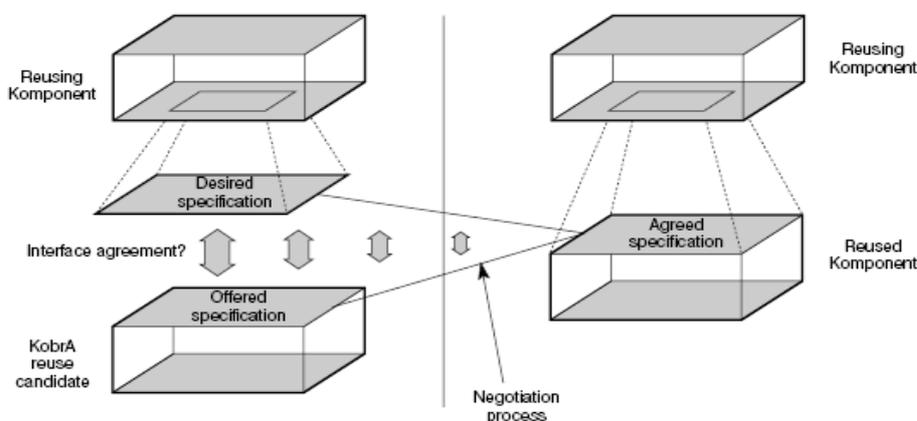


Figure 2.12: KobrA's reuse model [Atk02].

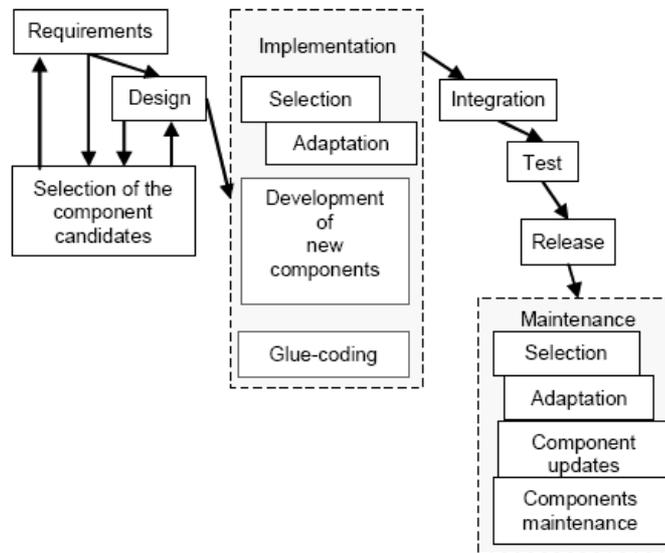


Figure 2.14: Waterfall-based reuse model [Crn06].

Because of the impact that existing components might have on the requirements the authors also advocate the consideration of reusable components already in analysis and design phases. The whole system development process is a constant trade-off between whether existing components are close enough to the requirements to be integrated economically. However, no guidelines are provided that explain how components can best be searched, and it should be clear that a pure specification-based retrieval as advocated by KobrA is not sufficient in this context. Rather, it might be desirable to have a more text-oriented approach that is able to find good candidates for a given requirement and to derive a kind of an “averaged” design from them.

3 COMPONENT RETRIEVAL

SO FAR

Good artists copy, great artists steal.
-- Pablo Picasso

From the overview of software reuse in the previous chapter it should be clear that neither component retrieval nor software reuse are new ideas and most of the underlying concepts are rather well understood. This chapter is intended to give an overview of what has been done in the area of component retrieval to date and to introduce our understanding of semantic component retrieval. In principle, component retrieval requires three prerequisites, namely a component repository where software assets can be stored, a representation format which is able to describe the assets concisely and a retrieval mechanism which is able to discover assets in the repository. These have been well characterized in the literature. The first problem is the so-called software *repository problem* [Sea99] which is about effectively collecting and storing a large number of software assets in a repository. The next problem has been called the *representation problem* by [Fra94] and deals with the issue of how best to represent software assets in a component repository. Finally, there is the component *retrieval problem* identified by [Mil98] which deals with the issue of finding the most suitable component retrieval techniques.

Previous researchers have proposed a variety of solutions for all three problems, but because of technical limitations none of them was particularly convincing in practice. Very recently, however, several fundamental technology developments have occurred which have the potential to radically improve this situation. These include the emergence of faster computers and larger storage devices, the wide availability of broadband Internet connections and the maturing of search-related open source tools such as the Lucene search engine² and the accompanying web crawler, Nutch. As we shall show in this thesis, these have opened the opportunity to create large-scale software repositories with improved search precision. Before we explain our approach for applying these technologies to support improved semantic search in chapter 4, in the remainder of this chapter we summarize the current state of the art in component retrieval.

²lucene.apache.org

3.1 SOFTWARE COMPONENT REPOSITORIES

As mentioned before, until recently, reuse repository systems suffered from three main problems, namely the repository problem, the representation problem and the retrieval problem. Typically, a search request to a (component) search engine starts with the user formulating a query that describes what he/she is looking for. The engine then transforms the query to its internal representation and tries to find “matching” results. In order to be included in the result set a candidate component has to fulfil the so-called matching criterion. Ideally, a search engine automatically ranks the results according to the closeness of the match, in other words it delivers the better matches first. However, fulfilling the matching criterion of the engine does not necessarily mean that a candidate component also fulfils the relevance criterion of the user [Mil98]. This is largely influenced by the latent conceptual gap [Lar05] between the concept the user has in mind and the actual software object as well as by the quality of the representation method and the retrieval algorithm [Fis91]. Thus, before this search process can be successfully applied, there clearly has to be a software repository that must be filled with an abstract representation of a set of components. Obviously, all three problems mentioned are closely linked together and a clever solution for the representation problem is a prerequisite for satisfactorily solving the repository problem and the retrieval problem as well.

3.1.1 COMPONENT REPRESENTATION METHODS

How to logically store software assets in a library is an aspect of software reuse that has often been neglected in the literature despite the fact that a repository’s component representation format determines the possible ways in which it can be searched. Even Mili et al., who presented a highly influential survey on “*storage and retrieval of reusable assets*” [Mil98], admit at the beginning of their article that there is little that can be said about the logical storage structure used in software libraries since the most commonly used structure is “*no structure at all*”. As it is practically impossible to define an isomorphic mapping from software to its functionality it makes sense to simply store software assets “*side by side*” (using Mili et al.’s terminology) in some kind of database. An obvious extension of this approach is the storage of additional information (i.e. metadata) about a component that should ideally be extracted by a tool without human interaction. Information about the programming language or signatures of a component’s interface are good examples of metadata that would allow more specific and faster searches in the collected data pool.

In their survey Frakes and Pole [Fra94] identified four basic representation methods. These are briefly explained in the following. *Enumerated classification* originates from library science and separates an area into mutually exclusive, typically hierarchical classes to create a taxonomy. *Ontologies* in the semantic web community [Ber01] might be considered a modern form of this approach. Typical problems with such an approach are the completeness of the taxonomy and the associated complexity that makes it difficult for humans to handle. Take for example the United Nations Standard Products and Services Code (UNSPSC), a common taxonomy targeting e-commerce products and services. It contains more than 18,000 entries and therefore far more than the prototype component repositories in the 1990s (and even some of today). *Faceted classification* [Pri91] and the slightly more general *attribute value classification* approaches are very similar and use a number of facets (resp. attributes) to describe an asset.

Each facet comprises a finite set of terms that can be chosen to describe the asset (e.g the programming language could be a facet). Each individual component has one of the allowable facet values for each distinct facet. In contrast, an attribute – such as the name of a component – can contain any arbitrary value. Finally, free text indexing approaches index textual information from an asset, i.e. the component or its documentation. It is surprising that most approaches in the past tried to use these methods separately from each other since today's web search engines, for instance, show that they can be easily and effectively used together. Thus, we believe it makes a lot of sense to use these approaches together to support searches on components. In fact, one of the research contributions of this thesis can be interpreted as an attempt to combine them in an optimal way, as described in chapter 5.

3.1.2 THE REPOSITORY PROBLEM

Relational database management systems have been around for quite a long time and are naturally suited for supporting the faceted and attribute value classification. However, only recent open source search engines such as Lucene are specialized on free text indexing and searching. Obviously, a combination of these both approaches would be useful, although, this is a non-trivial undertaking since a database normally lacks free text search capabilities whereas Lucene lacks relational data storage capabilities and thus up until now no solution has been published for this challenge. We will present possible solutions to this problem later in this thesis. Luckily, today's (software and) hardware systems are powerful enough to store large component indices of ten million or more components and carry out searches on them within less than five seconds as we will demonstrate later in this thesis. Thus, the repository problem in the sense of storing and quickly querying large component collections has been largely solved by the storage and processing capacities of modern computers.

Until recently, this was only one a minor problem anyway because there were not enough reusable assets to present a serious storage or searching challenge. Nevertheless, the question for the ideal size of a component repository has been another controversially discussed issue that has still not been resolved to date. Intuitively, it seems obvious that the larger a repository, the more useful it is for its users. However, some publications claim there is an upper limit on the ideal size although the goal of researchers has always been to create the largest possible component repositories. During the 1990s articles were published claiming the optimal size for a repository is somewhere between 30 and 250 [Pou99b] components and larger collections would unavoidably lead to degenerated content (e.g. out of date versions and descriptions etc.) in the repository. That opinion is interesting from today's point of view and can only be understood in the context of the relatively weak automated indexing systems that existed at the time. As a result, practical repositories had to be classified by domain experts as for example recommended by [Cal91]. Indeed, most successful implementations of component repositories at that time such as [Len87] or [Pri91] were around this size and can be regarded as manually built, centralized systems. Ironically, 30 or even 250 components would have been easy to browse manually or with the support of a simple keyword matcher and thus intensive research on retrieval mechanisms would have been superfluous. Perhaps this is the rationale for the claim in [Pou99b] that the retrieval problem could be seen as having been solved?

However, other researchers realized that the ever growing amount of reusable material on the early World-Wide Web provided an opportunity to automatically populate component repositories and tried

to develop search engines that automatically crawled for content on the web. The first attempt was initiated by the Software Institute (SEI) that developed the so-called Agora [Sea98] system in the late 1990s (see section 8.1.1 for a more detailed description). The idea was to avoid the huge upfront investment associated with centralized and manually filled repositories by filling them with components found by crawling the web. However, this attempt was unsuccessful due to a lack of hardware resources available for crawling and component analysis at that time. However, the situation has become even “worse” since large companies sometimes have hundreds of thousands of files in their version control repositories, but are typically not even able to perform simple text based searches over these resources let alone perform sophisticated semantic searches. With the advent of the open source movement a need for component search engines similar to common web search engines arose. And finally the standard libraries of common programming languages (such as Java) grew to several thousand components, not mentioning the large number of supporting frameworks containing tens of thousands of classes. This number can barely be handled by catalogue-based approaches as evidenced by the early Yahoo web portal. Such a large number of resources can only be managed by fully automated crawling technology and a sophisticated search solution as promoted by Google for about ten years now. Since the number of components in standard libraries and frameworks is already far beyond the above mentioned threshold for a centralized component collection, it is natural that there are already efforts under way to support developers in this “*API jungle*” in the form of “recommendation tools”, e.g. [Man05] or [McC07]. However, the heart of our problem is the large amount of open source software available on the Internet and in version control repositories of large companies. Although a large number of almost all kinds of systems has already been developed and published somewhere, it was virtually impossible to find a component that matched a specified design when the research for this thesis was started in 2004. The component repository systems of that time were neither able to index nor search such a massive amount of files. The recent commercial interest in code and component search engines shows that there has been (and still is) a growing need for better searches over components as well as for larger and more sophisticated component repositories.

3.1.3 USABILITY

Another important factor in the acceptance of a reuse system clearly is its usability [YeF05] and whether a developer has the feeling of receiving useful support or being bothered by a complicated reuse system that distracts him from his work [Fra95]. However, advances in hardware as well as the rise of platform-independent, integrated development frameworks such as Eclipse as quasi standards have opened up the prospect of proactive recommendation systems that constantly issue queries to component repositories in the background transparently for the developer. This contrasts with the traditional reactive approach where the developer has to trigger a search manually and consciously. The first examples of this kind of system were simple and context-free like Owen's [Owe86] “*Did You Know system*” (and not related to software reuse), descendants of which are integrated in many end-user products today. However, since the relevance of the delivered information in most cases was questionable context-sensitive systems were developed. Ye popularized a proactive component retrieval approach in his Ph.D. thesis [Ye01] where he developed CodeBroker, a recommendation system integrated in Emacs, a popular editor for the Linux operating system. CodeBroker suggests components based on names and comments extracted from the code a developer is typing. However, this system requires so-called “active commenting” in order to

create meaningful queries. McCarey et al. [McC07] recently presented a similar system called RASCAL for the Eclipse environment which aims to recommend useful method calls to a developer. More details about these two systems can be found in the section on related work. Although there has not been much discussion on this topic there seems to be a general consensus in the reuse community that a modern reuse system should be proactive and generate queries without any direction from the developer.

3.2 COMPONENT RETRIEVAL TECHNIQUES

Over the decades many different techniques have been proposed for retrieving assets from a software repository. Mili et al. proposed a classification of retrieval methods in [Mil98] which later made its way into the comprehensive reuse book [Mil02] by the same authors. According to this classification, we introduce the component retrieval techniques in order of increasing technological sophistication. This is to a large degree, identical to their chronological order of appearance. The given classification is not the only one in this area, and although it is not perfect from today's point of view it is by far the most comprehensive and is intended to provide a framework for discussion in this thesis. Additionally, to fully understand the ideas discussed in this thesis (and in this chapter) a number of general concepts from information retrieval and related disciplines are required.

3.2.1 INFORMATION RETRIEVAL

A large part of this thesis is about finding information that matches a user's need. Thus, it is necessary to introduce some foundations of information retrieval which is defined by Manning et al. [Man07] as follows:

“Information retrieval (IR) is finding material (usually documents) of an unstructured nature (usually text) that satisfies an information need from within large collections (usually on local computer servers or on the internet).”

It is interesting to mention that IR explicitly focuses on large collections and thus it is questionable whether the small component repositories of the 1990s discussed above deserve to be viewed as “component retrieval systems”. Information retrieval approaches usually index the terms found in a set of documents in a so-called term-document-matrix, i.e. they store the number of occurrences of each term per document. A nice overview of this topic is e.g. given by [Bae99]. A single document is represented as a vector, the so-called term-document-vector [Sal75], with one dimension per term (going out from all documents). Two documents can be compared with each other using vector similarity measures such as the cosine measure. In their simplest form the approaches merely use a boolean value for each term to indicate whether it is present in a given document or not. More sophisticated approaches store the number of occurrences for each term per document (so-called term frequency) or even multiply this value with the inverse number of occurrences over all documents (which is called inverse document frequency). This yields the so-called TFIDF (term frequency inverse document frequency).

These approaches purely operate on textual information and try to “interpret” their meaning with heuristics to deliver the information that a human would expect to receive for a given query. However, they are not able to recognize semantic relations between terms as recognized by a human. Furthermore,

problems usually arise when variant spellings, typos, synonyms (different words with the same meaning) or homonyms (identical word with different meanings) come into play. Various solutions have been proposed to cope with these problems, the use of stemming algorithms [Por06] to reduce words to their stem or the use of thesauri such as WordNet [Mil90] are some examples. Even techniques that are supposed to recognize semantic relations in free text have been developed [Dee90]. To compare retrieval techniques in IR, the concepts of recall and precision (see next paragraph) are typically used on a given and well-known reference collection. Approaches available today are able to achieve good results on collections of up to a few hundred thousand documents, but run into problems, such as a lack of precision or ever growing performance challenges, when the collections grow larger. As a result, it is difficult to implement the above approaches efficiently for web search engines for example, which often have to cope with billions of documents. We will discuss this issue more fully in the next subsection.

Recall and precision are accepted as the standard measures for the efficiency of retrieval mechanisms. Recall is defined as the proportion of all relevant documents that have been retrieved from a collection according for a given query and precision is the proportion of all retrieved documents that are relevant to that query. This definition makes one important assumption, namely, that the proportion of relevant documents in the collection is known a priori, an assumption which is unfortunately no longer valid for queries in web search engines [Lew06]. A formal description of the concepts is provided by [Bae99] for example. If R is the set of relevant documents in the collection of documents that should be queried, then $|R|$ is the number of documents in this set. Likewise if a retrieval system generates a set A as the answer to a user's request, then $|A|$ is the number of documents in A . RA is defined as the intersection of R and A - i.e. the intersection of all documents that are relevant and returned from the system and $|RA|$ is the size of the intersection. The following figure, adapted from [Bae99], clarifies this graphically:

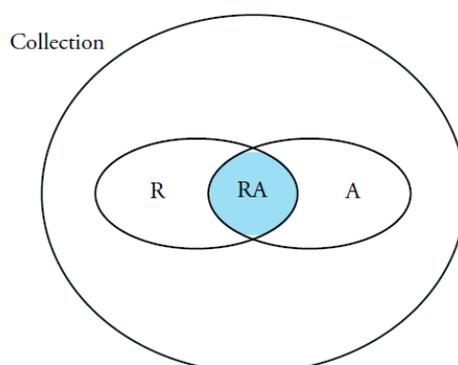


Figure 3.1: Illustration of recall and precision [Hum03].

Based on these definitions, recall can be written as $|RA| / |R|$ and precision as $|RA| / |A|$. Typically the user does not receive all documents that are considered relevant in one fell swoop, but in an iterative manner, one after the other, ranked by the degree of relevance. Hence recall and precision depend on how many of the most relevant documents are considered in their calculation. To depict the retrieval efficiency of an algorithm graphically so-called precision versus recall figures can be used.

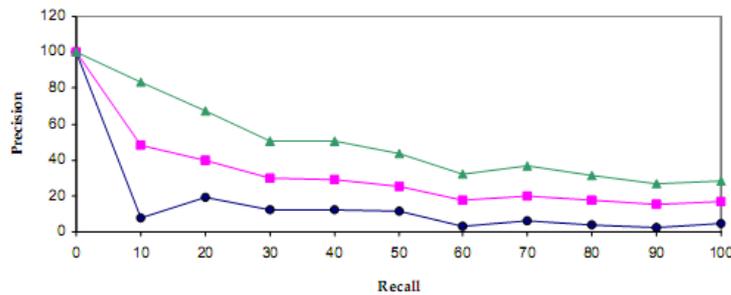


Figure 3.2: Recall versus precision curves comparing three different retrieval algorithms [Hum03].

The recall on the x axis is typically shown for eleven standard recall levels (0%, 10%, 20%, ..., 100%, interpolated if necessary) versus the precision on the y axis. Thus, the larger the area below the curve the better the retrieval algorithm. More details can again be found in [Bae99], for instance.

3.2.2 FOUNDATIONS OF SEARCH ENGINES

Since we will later rely heavily on search engines for component discovery we have to introduce some fundamentals of how search engines work. Before the advent of the World Wide Web search engines were not widely known. A few such systems existed in public libraries where they helped users to search for books according to keywords etc., but in general these systems developed by the information retrieval community did not make it into public awareness. The web, however, changed this situation fundamentally. Search and retrieval became vital to navigate around the large amount of unstructured data on the web. The first search engines (such as early Yahoo) relied on the catalogue principle and tried to manually categorize websites into a hierarchically organized collection. However, with the rapid growth of the web, this approach quickly lost ground against the “brute-force” crawl approach that Google and others have been using since the late 1990s. The basis for such an approach is, of course, an index of webpages, but a term-document-matrix as described above would require far too much resources. Thus, (web) search engines today rely on a so-called inverse index. In this approach it is not a list of terms for each document that is administered but rather a list of pointers to the documents in which each term appears. Although this principle does not allow sophisticated document comparisons like the application of the cosine measure, it has some significant advantages. The most important one is that for each potential search term there is immediate access to a list of documents containing it. Furthermore, this approach requires less data to be stored and can easily be distributed over multiple machines.

However, this idea was not very new and was not the reason for Google's rapid growth in popularity. It was the famous Pagerank algorithm [Pag98] that was responsible for Google's initial success. The motivation for Pagerank was to deliver the most important (and presumably the most relevant) websites for a query first. And the idea for a measure of popularity of websites in Pagerank is as simple as it is brilliant – just count the number of links that point to a site. The more pages with a high Pagerank pointing to a site the higher the site will be ranked. The calculation of “pageranks” for a considerable amount of websites is an iterative process, of course, but the values typically converge after three or four

iterations. Together with a few more tricks such as giving more weight to pages that contain the search keywords in the URL, title or headings, Pagerank was the foundation for Google's success. The Pagerank algorithm has already been adapted for use with software components by [Ino05] and a more detailed description of the calculation can be found in section 8.3.

As the main public search engines do not readily provide information about such things as their index size, there are few scientific publications on this topic and many of the estimates available on the web are only rough guesses. However, for the purpose of this thesis it is possible to get a coarse impression of the size and the capabilities of the main web search engines. As of 2005, it is estimated (or speculated) on websites about search engines³ that the major players like Google and Yahoo are able to index about 200 million pages per day which is about 1 to 2 percent of their estimated total index of around 10 to 20 billion pages. Assuming that the time for re-indexing is uniformly distributed for all pages this would mean that a page is re-indexed every 50 to 100 days. This is, of course, a rather long duration that would quickly lead to outdated indexes for pages that are frequently changed. This problem is recognized by the search engines as well, which is why they try to index pages that change often (e.g. websites of newspapers) more frequently. But even for the big players in the search business it is not realistic to create an index on a daily basis. This is an idea that Grub⁴ tried to implement around the year 2003. In the tradition of SETI@home⁵ that distributes the analysis of radio telescope data to volunteers that donate spare cycles of their computers, Grub distributed the indexing of the web to the computers of volunteers. However, the project had to be cancelled due to lacking resources.

3.2.3 COMPONENT RETRIEVAL APPROACHES

As stated by Mili et al., a retrieval process typically involves two criteria because a candidate component can fulfil the matching condition of one specific retrieval technique, but may not necessarily match a user's relevance criterion. For example, a keyword-based technique might retrieve 20 components matching the term "customer" but only 2 of them might actually fulfil the user's requirements for a customer object (perhaps the other 18 only have a reference to a customer object etc.) and thereby fulfil his relevance criterion. The authors divided the existing component retrieval techniques into the six classes shown below. We briefly summarize these techniques and the results of their assessment at this point and provide a more detailed overview of the retrieval methods later in this section:

1. Information retrieval methods
2. Descriptive methods
3. Operational semantics methods
4. Denotational semantics methods
5. Structural methods
6. Topological methods⁶

³ such as searchenginewatch.com

⁴ grub.org

⁵ setiathome.ssl.berkeley.edu

⁶ From today's point of view we prefer to view this as an approach for the ranking of search results.

Since component retrieval is a form of information retrieval it makes sense to reuse methods from the latter area to perform simple textual analyses on software assets. Descriptive methods go one step further and rely on an additional textual description of the asset like a set of keyword or facet [Pri91] definitions. Operational semantic methods rely on the execution or so-called sampling [Pod93] of the assets. Denotational semantics methods use signatures (see e.g. [Zar95] and [Rit89]) or specifications [Zar97] of assets while topological methods try to minimize the distance between the requirements and available assets based on a syntactic or semantic measure. Today, we would characterize these methods as a way to rank the results of a query. Finally, structural methods do not deal with the code of the assets directly, but with program patterns or designs. Overlap between these classifications can appear at various places, e.g. between (3), (4) and (6) as the behaviour sampling of components typically needs a specific signature or structure to work on. The authors provide the following table for each of the assessed groups according to a scheme with five discrete rates ranging from very low (VL), low (L) through medium (M) to high (H) and very high (VH). Unknown rates are denoted with (U).

Recall and precision have already been introduced as the two most important measures from information retrieval. The coverage ratio describes the average number of assets visited per query over the total number of assets in the library. Time complexity refers to an $O(N)$ measure for computation steps per query. In other words, low time complexity stands for a linear correlation, medium for polynomial and so on. Logical complexity refers to the power of the retrieval method in terms of predicates. In this context, very high means that second order predicates are possible. Finally, the meaning of automation potential should be obvious. The meaning of investment and operation cost should also be obvious, while pervasiveness reflects how widely a method is used in research and practice and the state of development ranges from a speculative idea to a fully supported industrial product. Again, the difficulty of use should be obvious while transparency describes the amount of knowledge a user of a method must have about the internals of the retrieval algorithm.

Method	Technical						Managerial				Human	
	Precision	Recall	Coverage Ratio	Time compliance	Logical compliance	Automation Potential	Inventory cost	Operation cost	Pervasiveness	Development state	Difficulty of use	Transparency
Information Retrieval	M	H	L	L	M	H	VL	L	H	H	M	H
Descriptive	H	H	VH	VL	L	VH	H	H	H	H	VL	VH
Operational Semantic	VH	H	H	M	M	VH	L	M	M	M	L	VH
Denotational Semanti	VH	H	H	VH	VH	M	H	H	L	L	M	M
Topological	U	U	VH	H	M	H	VH	VH	L	L	VH	VH
Structural	VH	VH	VH	VL	L	VH	L	L	L	L	VL	VL

Table 3.1: Assessment of retrieval methods according to [Mil98].

The authors conclude their survey with the following sobering statement:

“Despite several years of active research, the storage and retrieval of software assets in general and programs in particular remains an open problem. While there is a wide range of solutions to this problem (...) no solution offers the right combination of efficiency, accuracy, user-friendliness and generality to afford us a breakthrough in the practice of software reuse.”

In other words, where a technique offers sufficient precision it is usually too time consuming or too difficult to use or the other way round (i.e. easy to use but too many false positives or too few real positives at all). Hence, we believe a practical component retrieval engine requires a carefully chosen combination of various techniques from the above list (see section 5.2). To enable the reader to better understand the hybrid approach we propose later, we explain the existing techniques in the following subsections in more detail.

3.2.4 INFORMATION RETRIEVAL METHODS

The field of information retrieval (IR, [Bae99]) is much more mature and better understood than the field of software component retrieval. For example, the vector space model explained above, which is one of the most seminal retrieval techniques in this field, was originally proposed in 1975 by [Sal75]. The indexing process for such a system can be fully automated and enables the system to easily retrieve the most relevant documents for a query (which is represented as a vector as well) by calculating the cosine between two vectors. As stated above, since information retrieval is about finding information in libraries and software reuse is about finding software in software libraries it was obvious that ideas from the former could be helpful in the latter field. Moreover, it is clear that information retrieval methods can work by performing some kind of textual analysis of the text associated with software assets. These natural language elements can come from comments in source units themselves or text in analysis documents and user manuals. The latter, however, are not well suited for software retrieval since they are a relatively imprecise description of the source code. As [Mil98] ironically states, *“if traditional information retrieval methods were adequate in dealing with software assets, there would be little incentive to investigate other methods”*. Although not optimal for software reuse, research has shown it is indeed possible to regard software components or related documentation simply as documents containing information, even though this has serious drawbacks for precision and recall. This results from the fact that IR methods only extract textual information from the source code and use neither the syntactical nor the semantic information contained in it.

Nevertheless, IR methods have widely been used in various systems in the past due to their simplicity and the large degree of automation that is possible during the indexing process. Frakes and Nejme [Fra87] for instance presented a system that relied on extracting natural language information from header comments in C files. An important prerequisite for this method is of course the adherence to a coding standard and the user's familiarity with the behaviour of the retrieval system. More examples are given in [Mil98] where even a hypertext-based system was proposed by [Pou95] is reviewed. Two more problems arise in the context of text analysis – namely the synonymy and polysemy problems, the former arising from the fact that different words can have the same meaning while the latter describes the fact that one word can have different meanings. The so-called Latent Semantic Analysis (LSA, [Dee90]) tries

to extract concepts rather than just terms to damp these effects. This approach was used for instance in Ye's CodeBroker system [Ye01], but on a fairly small repository of a few hundred components. Although informal evaluations have provided good results, we do not believe the system would scale up since LSA is computationally very expensive and attaining an acceptable level of precision in the context of component retrieval requires so-called "active commenting", i.e. additional information that a developer has to provide. A more detailed discussion of Ye's system can be found in section 8.2.1.

3.2.5 DESCRIPTIVE METHODS

Similar to information retrieval methods, descriptive methods do not use the actual source code of a component, but additional metadata, i.e. typically a structured list of descriptive keywords. Mili et al. [Mil98] denote such descriptive methods as a subset of the information retrieval methods, but they give them their own category due to the high use of this approach in practice and literature. It is apparent that this approach is simpler to implement than the IR methods before since the component descriptions and searches typically only consist of terms from a controlled vocabulary. However, the indexing of components involves much more effort since this task is often difficult to automate and hence normally has to be performed by a human administrator. This implies a kind of natural upper bound for a repository using a descriptive method as it is not practicable to index millions of components in this way. Moreover, the administrator and the users of the repository have to have the same background or at least the same understanding of the vocabulary used to describe the components. In the best case this can simplify the retrieval of components, but in the worst case, new users have to become familiar with the description scheme before they are able to use the repository. Interestingly, websites such as del.icio.us have recently gained much attention with an approach called tagging where users can assign arbitrary keywords to websites and classify them therewith. Although, the vocabulary is not controlled this approach seems to work quite well as it can be used without learning the vocabulary in advance. However, a first project using Web 2.0 tagging for component retrieval reported rather disappointing results [Van06] compared with keyword-based retrieval.

The work of Ruben Prieto-Diaz, called faceted classification [Pri91], is a well-known example of the use of a descriptive approach. It was inspired from library science where systems like the Dewey Decimal [Cha94] provide an enumeration scheme with a finite list of predefined classes. This idea is very similar to the concept of ontologies [Ber01] nowadays proposed by the Semantic Web community. However, Prieto-Diaz argues that it is not always easy to select the class that describes a component best and hence relies on a faceted scheme of the kind used in library science since the late 1930. The term "faceted" in this context simply means that there is more than one way of classifying a component, for instance, as in Prieto-Diaz's example where a component might be described by several facets like design, program, structure, system etc. Other more practical facets might be the underlying component technology or the domain and so on. Such facets enable a much more concrete description of components and in Prieto-Diaz's system they are supported by a semantic network that defines a distance measure within a facet to enable (the most) similar components to be "recommended" when no direct match is possible.

3.2.6 DENOTATIONAL SEMANTICS METHODS

Mili et al. [Mil98] subsume both signature-based and formal specification-based retrieval methods under the notion of denotational methods. However, they had to accept that there is some debate on whether or not this is appropriate and argue that the former is only a subset of the latter. The denotational methods form the group of retrieval methods which is most appropriate for a design-based retrieval of software components as proposed for example by [Atk02]. Since the description of a software component typically consists of a syntactical description of its interface and a functional contract specification (cf. [Mey92]) it is natural to use these features to store component descriptions in a repository. However, this approach carries one inherent problem, namely that it is very difficult to formulate formal descriptions of components for queries and that an automated examination of the adherence to a formal specification is not possible due to the halting problem. Since this topic is closely related with theorem proving it is not surprising that denotational methods are most suitable for functional programming languages.

Signature matching, on the contrary, was considered the key to reuse by Zaremski and Wing in their eponymous paper [Zar93] from 1993. It is in fact an important prerequisite for both specification matching as well as for operational semantics methods, which we will discuss in the next subsection. Rittri was the first to suggest the use of signature matching for component retrieval in [Rit89]. As the name implies, signature matching originally focused on the signatures of functions. The author used the functional programming language ML for his research. The underlying idea of signature matching is to scan a component library for functions that have the same signature as the user's query, but to fully ignore function names. An exact match is achieved when the input parameter types and the return type of two functions match exactly without observance of the parameter names. [Zar95] defined a number of relaxed matches like permuted parameter order or even matches containing sub- or supertypes. However, to our knowledge only one publication [Str94] has tried to transfer these findings to an object-oriented language (Ada). This was not a simple undertaking since there is no sound type theory of the kind found in functional languages that would allow the definition of type isomorphisms.

Over the years many well-known approaches for the matching of formal specifications have been proposed. For example, [Per93] proposed a system containing predicates for functional features and interface descriptions while [Moi92] use algebraic specifications to describe the signatures and axioms of reusable components. On the other hand, [Jen95] introduced a system based on the formal specification of components and queries while Zaremski and Wing complemented their signature matching approach with work on specification matching [Zar97] where they use pre- and postconditions to describe components. However, the theorem proving required for assessing whether a component matches a query quickly became a bottleneck for practical implementations and the authors consequently tried to minimize the effort involved. [Pen99] proposed domain specific knowledge bases as a solution while [Fis91] introduced a stepwise filtering process to limit the amount of processing required. However, from today's vantage point these approaches all have limited usability since they require a human to create the specifications for each of the components in a library, a task certainly not feasible on today's libraries with millions of components.

3.2.7 OPERATIONAL METHODS

As the name indicates, this approach is based on the simple observation that software components have a dimension that other (i.e. typically purely textual) retrieval artefacts do not share: they can be executed and their reaction to given input stimuli can be evaluated. In other words, the component's behaviour is directly observable and does not have to be hidden behind some abstract description. The underlying idea is quite simple, the signature of the desired component is entered into the retrieval system together with some input/output pairs to be used for assessing the components. Each component in the repository is executed with the given input values and the output is compared with the expected output that has been fed into the system. Although appealing in theory, the approach has some practical limitations. First and foremost, components must be executable and although this sounds trivial, it sometimes is a serious problem to execute an individual class of a large project (such as Eclipse⁷). Moreover, side effects, non-termination, abstract data types and additional files that might be necessary to process results can cause further severe problems. Altogether, operational methods have essentially only been considered for functional languages where a sound type theory is available and signature matching [Zar95] is much better understood.

However, the first operational retrieval method, called Behaviour Sampling, was proposed by Podgurski and Pierce [Pod93] for simple C functions (i.e. functions that are free of side effects and only use simple variables) in the early 1990s. The main focus of their work was to estimate how precisely the approach worked for small sets of input samples, and four random samples were found to be sufficient in most of their experiments. Twelve random samples are considered to be the absolute maximum necessary for receiving unique results by the authors. However, as [Mye02] states, for software testing random value selection is a rather ineffective way of sampling the behaviour of components. Furthermore, the authors already realized that abstract data types must be broken down into their primitive parameters, a strategy that is known from algebraic specification.

Hall [Hal93] used user-selected samples (what we would today call test cases) to generalize the above approach and was able to retrieve not only simple components, but also composite components composed from other elements in the library. Complex data types could be retrieved through the use of constructors that merely contained simple variables. However his system neither supported polymorphism nor the isomorphism of signatures and is based on a functional language (Lisp). [Cho96] used finite state automata to model the behaviour of object abstractions, an approach that required a lot of manual effort since it could not be automated. Moreover, the use of internal attributes violates the information hiding principle and made it necessary to anticipate the implementation of a desired component in a query. [Atk95] defined a theoretical framework in Object Z that enabled a partial ordering of components in a lattice structure that in turn enabled component retrieval based on the most similar behaviour if no exact match could be found. However, no practical implementation of this idea was published at the time of writing.

⁷eclipse.org

3.2.8 STRUCTURAL METHODS

The retrieval methods introduced so far all try to approximate the functionality of a component in some way. Thus, the functional properties of a component are the matching criterion. The few techniques that fall into this category, however, have chosen a different way, namely similarity of the internal structure of a candidate to the component under consideration. [Mil98] argue that this technique is best suited to situations in which components have to be modified anyway after retrieval. In such a case it makes sense to look for a component whose structure is as close as possible to the structure of the desired component (look-alike instead of act-alike). Although this is the case with the other approaches as well, structural methods are supposed to be more suited in this regard. Mili et al. further argue that structural methods are especially well suited for white box reuse where the internal structure of components is available.

However, structural approaches have rarely made any impact on the practice of software reuse. Mili et al. have obviously also struggled to find meaningful examples for this category. There are only two main examples of attempts to apply these approaches in practice. One is the idea of programming clichés introduced by Rich et al. [Ric78] in the context of their Programmer's Apprentice project. Such clichés are somewhat similar to today's well-known Gang of Four design patterns [GoF95] although they are on the smaller level of idioms according to the pattern classification of Buschmann et al. [Bus96]. Structure is the basic selection criterion for clichés since they can be instantiated for a range of varying functions. The approach of [Pau94] is the only true structure-based retrieval approach published to date. Since structural matching requires the expected structure of the component under development to be defined, and this would normally be equal to programming the component, the authors define a higher-level language which is supposed to specify queries in a more abstract way. However, the authors see the main use of their approach in the context of program understanding and re-engineering which is perhaps why it would be difficult apply in a reuse context where one would have to anticipate the internal structure of the desired component.

3.2.9 TOPOLOGICAL METHODS OR RANKING APPROACHES

Topological methods rely on an underlying distance measure to find the component closest to a query. Consequently, it is obviously possible to not only deliver the closest result, but perhaps the ten closest results ordered according in the same way that WWW search engines select their search result today. Hence we prefer to view topological methods as essentially a tool that ranks the results of a query. In other words, topological methods must be built on top of at least one retrieval technique from the categories above that can be measured in some concrete way. For some, such as the information retrieval approaches, this appears to be straightforward since the frequency of terms could be counted, for example. But even for the signature of a component it is possible to define a distance measure such as the number of steps necessary to transform one signature into another (cf. [Kra03]). Unfortunately, this would be very expensive to implement in a search engine since, in principle, the distance from each query to each entry in an index would have to be calculated in each search.

Girardi and Ibrahim [Gir94] developed a system which is often misinterpreted as a way of extracting linguistic, syntactic and semantic information from reusable artefacts and their documentation in order to deliver components that are as close as possible to user requests. However, they merely experimented

with the extraction of syntax and semantics from textual information and not from source or binary artefacts. Thus, they applied a purely textual information retrieval approach where they calculated a distance between query and candidates. The authors evaluated their system on an index composed of a few hundred Unix commands. For their natural language queries they reported an average recall of about 0.99 and an average precision of nearly 0.90 based on twenty queries using the index created from the “man pages” of the Unix commands.

Mili et al. list a few other approaches that operate on different underlying distance metrics, but adhere to the same principle. The term “ranking”, however, is not used in this publication. The idea of ranking the results of component searches was, to our knowledge, first introduced with the work of [Ino05] (a more detailed description of their work is provided in section 8.3) that realized that a simple search and retrieval approach is no longer sufficient for “larger” repositories. They were inspired by Google's Pagerank algorithm [Pag98] and based their ranking not on the closeness of the query to the candidates (they only use a simple keyword matching for this), but on the popularity of the candidates. In other words, during the creation of its index, their system extracts how many classes use another class and the more popular a class the higher it is ranked in the set of results. A similar approach is used in the Sourcerer project [Baj06] from UC Irvine. The disadvantage of such an approach, however, is that it will only work with known collections in which a naming scheme is applied and no duplicates or similar (e.g. older) versions can appear. Currently, it seems unlikely that this will ever work with data from the web or from unknown open source collections.

3.2.10 DISCUSSION OF CLASSIFICATION

The classification of Mili et al. is certainly a valuable tool to distinguish the various groups of reuse techniques. However, in our view its focus on reuse techniques is its main weakness in the context of component-based reuse. The authors created a generic scheme applicable for all software assets that might be reusable during the development process. Thus, it is not very descriptive from the point of view of components. More specifically, it combines some issues that should be separated in one group (e.g. it combines syntactical and semantic aspects into denotational techniques) and separates some issues that should be combined into different groups (e.g. the distinction between structural issues and syntactical issues in the denotational techniques).

Consequently, we suggest a more component-oriented classification, ideally inspired by modern component-development approaches such as Kobra [Atk02]. As we pointed out in section 2.4.1. Kobra defines a black box view on a component, called a specification, which typically comprises three views of the component, namely a structural, a functional and a behavioural view. While the structural view captures the syntactical aspects of a component and its environment in UML class diagrams, the latter two perspectives contain semantics information in terms of operation specifications and the externally visible states of the component. Kobra also strives to offer a concise but minimal description for components. Given this model, we believe it makes sense to classify component retrieval techniques in a similarly way and propose the following groups:

1. structural (or syntactical)
2. functional semantics
3. linguistic

The structural techniques comprises all approaches that try to match components based on their structural properties, this includes pattern-based approaches [Ric78], signature-based retrieval [Zar95] as well as more recent attempts to perform retrieval based on UML class diagrams [Llo04]. The group of semantic techniques includes everything that deals with descriptions of the functionality of components and their behaviour. This includes static (e.g. specification matching [Zar97]) as well as dynamic techniques (such as behaviour sampling [Pod93]). Although the linguistic aspect is not mentioned in the development context of Kobra it has to appear in the retrieval-oriented context of this thesis. Although all component descriptions contain names and other linguistic elements these do not guarantee any behaviour and can totally be free of meaning (e.g. “sdfsdfs” would be a valid name for a class). However, in an object-oriented system they normally give valuable hints for the purpose of objects. The groups of retrieval techniques mentioned above thus are orthogonal and capture different aspects of components. None of them would be sufficient to describe a component fully on its own. For example, as a recent publication [Kra03] has shown the interface of components holds some information about its functionality, but is not sufficient alone because it does describe the full functional semantics.

Although the three perspectives from above cover the functional aspects of a component, they do not include the non-functional quality aspects. These were not explicitly considered by Mili et al. Either, but are also important and can be used for the ordering of components within a result set. Given the different approaches currently known, we propose to classify approaches for ranking components into the following groups –

1. quality of service
2. popularity
3. distance to the query

The first two groups can normally be applied independently from the query and thus can be calculated at index creation time. Since they do not influence how well a component matches a query they should only be applied on groups of results which have the same degree of match to the query. The third group is different since it ranks components according to their distance to the query. Consider, for example, a component that has four operation signatures matching the query. It is obvious that it should be ranked higher than another component matching only two of the signatures.

Finally, the core challenge from a reuse point of view is to find a simple and comprehensive representation of components that allows simple query formulation for the user without the detailed knowledge of a new query language. Up to now, the opposite has been the case. Query formulation has largely been driven by the selected retrieval technique and hence has ranged from simple keyword searches to the formulation of formal specifications. However, as pointed out in the introduction, these approaches are often unnatural for developers normally working with UML diagrams or source code.

3.2.11 RETRIEVAL TECHNIQUES IN USE TODAY

The advent of web-based code search engines has brought some new component retrieval techniques into focus. Thus, we briefly provide an understanding of the techniques that are in use today and that will appear frequently in this thesis. We start with a structural retrieval technique. We will use the term *signature-based retrieval* in accordance with the signature matching approach [Zar95] mentioned above. Consider the following class diagram of a small `Stack` component as an example.

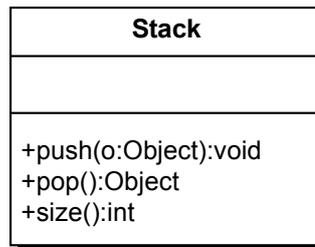


Figure 3.3: Class diagram of exemplary `Stack` component.

From this information, signature-based retrieval would use the following signature for a search:

```

Object -> void
void -> Object
void -> int
  
```

The simplest linguistic approach still in use today is of course plain *keyword matching*, whose relevance criterion is just the appearance of the required keyword somewhere in the candidate component, i.e. typically in the source code. This would result in the following query for the above example where all names and parameter types have been extracted from the component's interface:

```

stack push object void pop size int
  
```

A number of the structural techniques that have appeared recently have also been influenced by linguistic approaches. The simplest one is the so-called *name-based retrieval approach* that limits keyword matching on specific structural elements of a component, i.e. typically the method and class names:

```

stack push pop size
  
```

The so-called *interface-based retrieval* approach uses the complete interface information contained in the UML diagram, and the search engine is expected to recognize them properly. Queries are typically expressed in a special query language, such as the UML-like one that we developed for this thesis, –

```

Stack (
    push(Object) :void
    pop() :Object
    size() :int
)
  
```

or directly as (Java) code as in the following stub:

```
public class Stack {
    public void push(Object o) {}
    public Object pop() {}
    public int size() {}
}
```

Previously, *specification-based retrieval* meant retrieval based on a formal specification of a component such as the following excerpt of a Larch/ML specification for a Stack presented in [Zar97].

```
signature StackObj = sig
  (** using OrderedContainer **)
  type  $\alpha$  t (** based on
    OrderedContainer.E OrderedContainer.C **)

  val s :  $\alpha$  t

  val push :  $\alpha \rightarrow unit$ 
  (** push (e)
    modifies s
    ensures  $s' = insert(e, s)$  **)

  val pop_top : unit  $\rightarrow \alpha$ 
  (** pop_top () = e
    requires not(isEmpty(s))
    modifies s
    ensures ( $s' = butLast(s)$ ) and
      ( $e = last(s)$ ) **)

  val height : unit  $\rightarrow int$ 
  (** height () = i
    ensures  $i = size(s)$  **)
end
```

Figure 3.4: Formal specification of a stack
in Larch/ML as given in [Zar97].

However, since we intend to use this term for retrieval based on a (KobrA) specification of a component as defined in the last subsection, we prefer to call the former *formal-specification-based retrieval* to distinguish the two.

3.3 SEMANTICS IN REUSE APPROACHES

Few if any of the retrieval techniques developed to date have fulfilled their requirement. As has been explained during this chapter, most attempts have only been prototypical implementations which have demonstrated features in carefully controlled environments “in vitro” and never “in the wild” (or “in vivo”). Furthermore, none of these approaches could really be called “semantic”. Semantics, defined as “*the study of meanings*” by Merriam Webster’s dictionary, is a term that is connected with components in a variety of ways. There is a textual semantics dimension related to the meaning of names in source code and perhaps also in the documentation of components. As shown above, this is typically used by approaches based on classic information retrieval techniques. Unfortunately, names or text in a component do not necessarily determine any of the behaviour or the functionality of a component although one fundamental rule in object-oriented design is to keep the so-called “conceptual gap” between an object in the real world and its software representation as small as possible [Lar05]. Assumed that this rule is adhered to, it is likely that the textual semantics are helpful in getting a first idea of the

domain a component is used in. However, it is also likely that in large repositories such an approach alone will not be precise enough as our results shall indicate later in this thesis.

This is one of the reasons why the semantic web community has been trying to enrich for example web pages and web services with ontologies that can be used for reasoning within service repositories for automated discovery, matching, and composition. However, practically usable implementations are still a long way off and some concerns that the research has lost its focus and is drifting away from this goal have been raised [Shi07]. The main problems arising in this context are the complexity of the ontologies created and the problem of automatically matching different ontologies with each other. Furthermore, there have been attempts to use the formal semantics of components for retrieval [Zar97], but these are typically too complicated to use. The idea is based on the design by contract approach [Mey90] and uses pre- and postconditions to capture the behaviour of operations. However, formal methods are not very popular amongst developers since their use is typically as complex as programming the actual solution. Additionally, a concrete mapping from functionality to formal description must be developed by hand for each component appearing in a repository and there is not even a chance to automatically check whether a component adheres to its formal specification due to the halting problem. The above mentioned operational semantics methods [Pod93] have tried to circumvent this problem by directly observing the behaviour of simple components, stimulated with a few random values. In our opinion, this is a promising approach. However, as we discussed above has not been sufficiently developed and investigated for modern programming languages.

For the purpose of our dissertation we define semantic component retrieval as the ability of a search engine to deliver the components that have the lowest conceptual gap to the software object that a developer has in mind. While this might at first sight might appear to require mind-reading, it actually means nothing more than optimizing a retrieval system for various conceivable component search use cases under different circumstances. Since there has to date been no description of such use cases in the literature we shall identify and explain the use cases in chapter 5 where we describe the core of our semantic search approach.

4 THE INTERNET AS A REUSE REPOSITORY

Getting information off the Internet is like taking a drink from a fire hydrant.

-- Mitchell Kapor, Lotus Corporation

The main technical obstacles to widespread, systematic reuse have remained the same ever since the idea was first put forward – how and where to find components suitable for a particular need in a particular context. One obvious approach is to create a component repository which component suppliers and consumers can use to match their needs and services through universally agreed categorization and description rules, as described in the vision for UDDI in section 2.4.2. However, while there have been serious efforts to create practical component repositories along these lines, as Seacord concludes [Sea99], there will not be a “*useful solution to the software repository problem without education and direction from a central group advocating the establishment of these software engineering repositories*”. Still, from today's vantage point, it seems unlikely that such a centralized approach will ever make a significant impact on commercial reuse levels any time soon. This view was reinforced by the quiet shut-down of the UDDI business registry (UBR) in early 2006. The UBR was set up with huge upfront investment by IBM, SAP and Microsoft as a showcase for a worldwide service repository. However, as our investigations [Hum06] shortly before the shut-down revealed, it contained very little usable material and the effort put into its creation never paid off.

Right now, it is difficult to judge whether the similarities between evolution component/service search engines and general web search engines are merely coincidence, but it is interesting to note that Yahoo (like the UBR) also started as a browsing-oriented catalogue of web pages which relied on the entries of users. Obviously, the web grew too fast and Yahoo finally also switched to a crawling-based approach. Therefore, we conclude that the most promising way of promoting component-based reuse in the foreseeable future is to find better ways of automatically using the largest and most widely accessible knowledge and software base in the world today: the Internet. Indeed, the arrangement of data on the Internet is obviously the antithesis of a strictly organized repository. Information on the World Wide Web (WWW) is organized in a large variety of different ways, with no central control or standardization other than at the protocol level. The WWW is the part of the Internet determined by the use of the

Hypertext Transfer Protocol (HTTP) and has made its way from an underestimated research project [Ber99] into our everyday life bringing along three major advantages as an information resource:

1. it vastly overwhelms any other repository in terms of scale and content
2. it is freely available and
3. it is the focus of the most advanced searching tools available today – namely web search engines such as Google etc.

Thus, it makes sense to consider the use of the Internet as a source for reusable components, especially since many of the techniques that haven't proven successful in this disorganized environment should be applicable in the context of company internal projects as well. Still, as mentioned before, keyword-based searching is the preferred way to find information on the Web today. Search engines like Google, Yahoo, Lycos and others are currently the most sophisticated tools used to find information on the web. Recent estimates such as that from [Gul05] give 11.5 billion indexable pages as a lower bound for the size of the Web in 2005. Google, for instance, claimed at that time to have more than 8 billion pages indexed. It is likely that amongst those is a large number of source files and Java applets from the web and even the CVS and SVN servers of a lot of open source repositories are accessible through the Internet.

When work on this thesis was started in 2004, the idea of using the Internet as a reuse repository was already more than five years old. In the late 1990s the Software Engineering Institute (SEI) had tried to crawl the web for reusable Java applets using special queries to a mainstream search engine. However, the experience with their so-called Agora project [Sea98] was disappointing since their system was not able to effectively process the amount of data created by that endeavour. More information on this project can be found in section 8.1.1. Fortunately, times have changed since then and in the last five years the technological environment has become much more favourable. As we shall explain in this chapter, the Internet has indeed become a valuable source for software components. In the following, we try to assess the theoretical potential of the Internet as a reuse repository and to estimate the number of source (and binary) files available from it. Furthermore, we discuss, how general style search engines can be used for targeted source code searches and what material we were able to find for the Merobase search engine that demonstrated that it is possible to gather and populate a dedicated component search engine with content found on the Internet. We round this chapter off with a discussion of the difficulties involved in publishing components on the Internet in order to make them findable.

4.1 ESTIMATED POTENTIAL

One way of estimating the number of available files on the web is of course to use web search engines to search for appropriate files. We developed some simple heuristics that enabled us to yield very precise results for most programming languages, at least from the two biggest search engines Google and Yahoo. Interestingly, [Yao04] still denied the feasibility of such an undertaking in 2004. Our basic idea is to use some undocumented features of the filetype filters of the two identified search engines to constrain searches to files in a desired programming language. For example, it is possible to restrict searches to Java files with *filetype:java* in Google queries and with *originurl:extension:java* in Yahoo queries. Adding for example “*class stack*” to such a query will deliver stack components with a surprisingly high precision.

Further heuristics to search for operations etc. are presented in the next subsection. The heuristics can also be used to estimate the number of files in a given programming language that are indexed by mainstream search engines and hence allow to estimate at least a lower bound for the total number of such files available on the web. To illustrate the magnitude of the accessible code resources on the web, table 4.1 shows the numbers of Java files that could be retrieved using the Google and Yahoo search engines during our experiments over the last years. Two sets of values are shown for the Google entries – the first giving the number obtained using the regular human HTML interface and the second (bracketed) giving the number obtained using the web service API for automated access. Unfortunately, the latter delivers only a fifth of the results available using the former and is no longer supported. This, of course, makes it less appealing to use the Google API for issuing such metasearches.

Month	Google (Web API)	Yahoo
08/2004	300,000	-
01/2005	640,000	-
06/2005	950,000 (220,000)	280,000
08/2005	970,000 (220,000) <i>1,510,000 (367,000)</i>	2,200,000
11/2005	2,210,000 (190,000) <i>4,540,000 (410,000)</i>	2,200,000
03/2007	1,350,000	470,000
06/2007	2,900,000	680,000
11/2007	1,300,000	1,000,000

Table 4.1: Number of Java files indexed by search engines on the web.

The italicized values in the fourth and fifth row stem from the query *“filetype:java” class OR – class* that – strangely enough – delivered significantly more results for Google than just *filetype:java class*. One would assume that a search with *“filetype:java” -class* would only deliver Java interfaces and no classes but actually, this is not the case. Manual inspections revealed a high percentage of class files. One explanation for this strange result may be that Google does not completely index some files. Furthermore, Google obviously changed its system in 2007 so that a plain search for *filetype:java* started to deliver results without further search terms (which did not happen before). The numbers in the table represent the mean value of samples per month whereas individual values can vary even from one request to the next within just a few minutes. However, the growth trend illustrated by the numbers is unmistakable even though the numbers also show the sustained effort of Google and Yahoo to clean their indices from files not containing natural language. In August 2005, similar requests for various C-style languages (filetypes: c, cpp and cs) revealed a total of about 1.6 million source files in Google’s index, and 2.7 million from Yahoo.

The overlap between Google and Yahoo seems to be rather low - it is typically below 20%. For example, only 5 out of 24 results for the `isLeapYear` example used in chapter 7 were the same and in the first 250 results of each engine for the `Matrix` example from the same chapter, only 47 out of 500 overlap. This observation tallies with other reports for general HTML searches as those described in [Dog05] for example. Additionally, it is interesting to observe that both search engines were apparently surprised by the massive growth of open source software on the web in 2004 and 2005. At that time, both engines also indexed source code from the WebCVS (or -SVN) interfaces of the large open source hosting sites, but as of 2007 both engines have obviously removed these files from their indices. This is not surprising since indexing source code is not the goal of commercial search engines as Peter Norvig (head of search quality department) from Google confirmed in a private e-mail conversation in 2005. The observable decrease in the number of Java files on Google by about three million Java files after that operation corresponds pretty well with the amount of files that our own crawling efforts for the open source hosters delivered in 2006 (cf. table 4.6). In summary, we estimate that about 3 million Java source and about 2 million C, C++ and C# files (without WebCVS/WebSVN) are available on the open web at the time of writing. Additionally, at least 3 million Java and 2 million C language source files are available in open source repositories. Due to the rapid growth of both the web and the open source community, it is likely that these numbers will grow steadily in the next few years, even if the millions of code snippets embedded in an uncountable number of web pages are not taken into account. However, the large number of exact or near duplicates (some files appear more than a dozen times in our index) makes it even harder to find a good estimate of reusable components on the web. Although the exact number of software components is indeterminable and in constant flux, the following fact is clear – the number of components available through the Internet exceeds every pre-millennial component repository reported in the literature by at least three orders of magnitude.

Google and Yahoo might also be helpful for the web service community since they are also able to retrieve WSDL files. As the next table illustrates, the number of files is high compared to the values for the former UBR presented in [Hum06], but more a detailed validation has shown that most of these WSDL files are not backed up by a working implementation of a service.

Search Engine	API	Claimed no. of links to WSDL files	No. of actual links to valid WSDL files
Google	yes	9000 (1700)	794 out of first 1000
Yahoo	yes	13400 (1900)	425 out of first 1000

Table 4.2: Number of WSDL files delivered from search engines.

The values in brackets show the number of results returned through the APIs. This indicates that the search results could be better if the artificial limitation on automated queries were removed. Both search engine APIs allow automated access to only the first 1,000 results returned in response to a manual query. This is usually not a problem when searching for a specific functional component since the number of retrieved candidates for a specific query rarely exceeds a few hundred.

Given the total numbers presented in this section, we can estimate that roughly 1/1,000 of the pages indexed by the two big search engines were source files in August 2005. After the exclusion of CVS and SVN content this number seems to have dropped to about 1/10,000.

4.1.1 SPECIALIZED SEARCH ENGINES ON THE WEB

As of 2004 when work on this dissertation was started, no specialized code or component search engine were available on the web as we have already pointed out above. But since then the situation has changed considerably and four serious commercial projects emerged. In order of appearance these are:

1. koders.com, started in late 2004 by a development company in California. Koders was the first code search engine on the market, focusing on components from public CVS servers.
2. krugle.com, backed up by several million dollars of venture capital, Krugle started in late 2005 with a beta version but needed until June 2006 to offer public access for everybody.
3. Merobase.com is the search engine that emerged from this dissertation and went live in July 2006.
4. google.com/codesearch. Google followed with its code search engine in fall 2006.

In contrast to general web search engines the named sites are specialized for source code searches. Hence, they all offer the opportunity to limit searches to a specific programming language, and they all fulfil another important requirement for being accessible by external tools – namely they provide an API for programmatic access. The APIs are based on Amazon’s Opensearch format [Cli07] which in turn is based on RSS. When estimating the size of their repositories by counting the number of Java classes (by searching for the terms “class” or “interface” in Java files) we found the last three engines having more than 10 million components in various languages in their indices as of November 2007. We will give a comprehensive overview of known code search engines and their content when discussing related work in section 8.1.2. More details on the content indexed in the Merobase search engine can be found in section 4.4. In section 4.3 we provide more details on its capabilities and on, the different types of components that it contains and other interesting facts learned during its construction. In a nutshell, we currently have about 4 million Java source files indexed in Merobase, 3 million originating from open source hosters and about 1 million from the World-Wide Web.

Since one of the reasons for the recent excitement around web service technology was its search capabilities (UDDI [New02] was supposed to bring together service providers and service requesters) we continue our overview with an analysis of web services repositories and the services that they offer for third-party (re-)use. UDDI used to be (and sometimes still is) advertised as a flexible brokering technology that allows component developers to “publish” their software as services, and potential component users to automatically find suitable services via formalized syntactic descriptions of their requirements (in the form of WSDL documents). Even semantic composition capabilities for web services are becoming available (e.g. with the help of OWL [Ant04]). Since so much industry investment had been pumped into the UDDI Business Registry (UBR), one would have expected a sizeable index of services to be available. However, as table 4.3 demonstrates, the UBR (and other service repositories) failed to reach a critical mass of entries and a large proportion of the entries contained in the repository were out of date. Many entries did not even point to valid WSDL descriptions and of those that did,

only a small proportion were actually backed up by working implementations. The UBR's shut-down in early 2006 was a logical consequence.

Search Method	API	Claimed number of links to WSDL files	No of actual links to valid WSDL files
UDDI Business Registry ⁸	yes	770	400
BindingPoint.com ⁸	no	3900	1270 (validated)
Webservicelist.com	no	250	unknown
XMethods.com	yes	440	unknown
Salcentral.com ⁸	yes	~800	all (validated)

Table 4.3: Number of WSDL files within reach at various websites (July 2005).

However, the main problem with the UBR's concept in our opinion was not a technical one, but the overhead involved in the manual creation and maintenance of the repository. The effort involved in entering a complete service profile into the UBR should not be underestimated. In addition, the effort involved in updating or removing the (possible many) entries when a server was moved or closed down should also be taken into account. In theory, this should have been taken over by the publisher who entered a service in the UBR, but this is often forgotten in practice. Interestingly, the UBR followed exactly the three-phase reuse progression (empty, filled with little content or filled with a lot unusable content) that Poulin reported in [Pou95] from his practical experience at IBM (although we would argue that the UBR actually never reached the third phase). In general, we can only speculate about the reasons for the disappointing performance of such repositories. One feasible explanation is the simple fact that there were not many services available at that time. We were able to discover about 3,000 working services in 2006 for Merobase and only recently another web service search engine (seekda.com) that emerged from an EU-funded project has been able to collect more than 10,000 publicly available web service endpoints. Our efforts in late 2007 also led to about 12,000 such service endpoints.

In addition to these code and web service search engines, there have been numerous attempts to establish commercial component "marketplaces" in recent years. However, these have also had only limited success. Two of the most well known, ComponentSource.com and Flashline.com, had to merge in 2005. Moreover, the UDDI Business Registry (UBR), the high profile industry repository for web services contained very little useful material (as we will show later) and was finally shut down in January 2006⁹. Likewise, most other initiatives have had very limited impact. These approaches have essentially all been based on a standard "e-retail" model in which components are offered in an informal catalogue-like style as if they were mainstream consumer products. Trying to discover a component at ComponentSource is therefore still much like browsing for a book on Amazon. It is a very informal, unpredictable process with a highly uncertain outcome. Of course, searching tools are provided, but these are very simple,

⁸ As of 2007 this website is no longer available.

⁹ The official rationale is that the UBR has been successful as a proof of concept, though.

typically text-based technologies that essentially look for keywords in a component's documentation. They are still far away from a semantic matching or at least a matching based on the signatures (i.e. the parameters of methods) or the full interface (i.e. parameters and names of methods) of classes.

4.2 PRECISE RETRIEVAL WITH GENERAL-STYLE SEARCH ENGINES

Although there is a number of specialized code search engines around, “metasearching” general search engines can be an appealing option to find software components on the web. No special infrastructure involving potentially thousands of computers is necessary to answer queries, for instance. It is sufficient to offer a more specialized user interface and to send the actual query to one of the large general search engines. There are probably hundreds of thousands of developers that use the web on a daily basis to collect reusable source snippets or to draw inspiration from open-source software and who would welcome such a search engine. But, “abusing” general-style search engines like Google for software component searches is not simple and some researchers like Yao [Yao04] doubted that this would be possible at all while others tried it in an unsanctioned way [Ino05]. However, researchers from various areas have been using queries enriched with special keywords on general-style search engines for many years. For instance, we have been using this approach with Google to extract information on music perception in 2003 [Bau05] from music related websites and [Ino05], as just mentioned, enriched searches with the terms “java” and “source” to limit results to these kinds of files.

However, although queries of this form deliver pages that may well contain information on the topic desired, the hit ratio for actual source code is still rather low. Fortunately, as briefly indicated above, a better way is of doing this is provided by at least Google and Yahoo. After studying the advanced features of today's two most important search engines, we were able to develop some simple heuristics that can be used to limit searches to a specific programming language [Hum04] and even to retrieve classes or methods with a high precision. Both engines offer a filter that limits searches based on the “filetype” of a web page. Officially, types like pdf or doc are supported that contain textual information that might be interesting for people to read. Unofficially, however, file extensions of common programming languages like java, c, or cpp are also supported although in some cases the filter does not work perfectly (e.g. for links to CVS pages that end on “.java” but contain HTML content). But in general, the pureness of the results is larger than 95%. The following examples show how to use this feature to estimate the number of java files within the indices of Google and Yahoo:

Google: filetype:java

Yahoo: originurlextension:java

Both forms are simple, but unfortunately most of the commercial search engines limit the number of results a user can access to avoid too high a processing load and the undesirable “exploitation” of their indices. Google and Yahoo display the estimated total number of hits although only the first 1000 results can actually be accessed. However, whenever an API for automated access is available, such exploitation cannot be totally avoided since it is possible to enrich search requests with further terms from a dictionary to get more results as Seacord has already demonstrated with Agora [Sea98], the SEI's

software retrieval engine (see also section 8.1.1). Hence, the number of requests that can be issued via the search engine's API is typically also limited to a few thousand requests per day.

Going back to our heuristics, a simple “speculative” search for a stack component might look as follows:

Google: filetype:java stack
Yahoo: originurlextension:java stack

The drawback of this approach is that any source file that somewhere contains the term “stack” will be retrieved no matter whether it appears in the component's name, in identifier names, strings or documentation. Hence, a better heuristic to limit searches to a specific class, i.e. a stack in this case, has the following form:

Google: filetype:java “class stack”
Yahoo: originurlextension:java “class stack”

To attain even higher precision it is necessary to search for operation signatures as well. However, this is difficult in this context since methods in Java have no keyword that could be used to filter the results. The simplest possible heuristic in this case is to add the method names and perhaps the parameter types as follows:

Google: filetype:java “class stack” “void push int” “int pop”
Yahoo: originurlextension:java “class stack” “void push int” “int pop”

This is possible since special characters like brackets etc. are ignored by the search engines. Suppose we are looking for a method that has more than one parameter. It is highly unlikely that we would find anything if we have to specify the parameter names since in Java, these are listed between the types. However, Google (and recently also Yahoo) have limited support for the asterix character as wildcard and hence support the following kind of query:

Google: filetype:java “int add int * int” “int sub int * int”
Yahoo: originurlextension:java “int add int * int” “int sub int * int”

These simple examples show how, given a precise knowledge of the required interface, ordinary search engines can be used to discover source code components with a very high precision. Of course, the recall tends to decrease the more operation signatures are added as more and more signatures and their orders have to be anticipated correctly. However, our initial prototypes have shown that this approach is well applicable to reduce the number of candidates to a reasonable amount, which can be processed further afterwards.

4.2.1 (META-)SEARCHING WITH GOOGLE CODESEARCH

More than two years after our initial experiments with the general version of Google's search engine the company released its own dedicated code search engine. Compared with other engines that were around in late 2006, its interface looked rather premature. However, it offers an API for programmatic access and possesses one of the largest indices of code currently available. This makes it interesting for

metasearches. Although it does not support any kind of interface-driven searches, it was the first code search engine to support regular expression (“regex”) searches. This enables a more efficient filtering process since regex are more powerful than the primitive wildcard character mentioned above. Hence, we developed some regular expressions that are able to describe the signature of components in Java-style programming languages. The general idea is similar to the one presented above, keywords like method names or parameter types and other fixed elements like brackets are extracted from a component's signature and other variable elements that are required for a text-based search on Google Codesearch (like e.g. the parameter names) are replaced by regex constructs. For example, a parameter name in a method signature could be replaced by the following regular expression: `[a-zA-z0-9]+`

This means that a parameter name must contain at least one upper- or lower-case char or a number. This technique can be applied for all signatures in a component and can mimic, to a reasonable extent, signature matching [Zar95] in specialized search engines. Take for example the following regex query that could be derived from a `Customer` object with `getAddress` and `setAddress` methods:

```
(class\s+Customer[\s+|{|}|\s+(program|unit)\s+Customer)

(String\s+getAddress\s*\(\s*\)|((procedure|function|def)\s
+getAddress\s*\(\)\s*:\s*\s*String)

(void\s+setAddress\s*\(\s*String\s+[a-zA-z0-9_\$]+\s*\)|((procedure|
function|def)\s+setAddress\s*\(\s*[a-zA-z0-9_\$]+\s*:\s*\s*String\s*\)\
s*:\s*\s*void)
```

However, as becomes apparent by this example, regex can quickly become complicated and even experienced regex users are likely to make errors in defining such expressions, especially when they have to be formulated in Google's small query box without any syntax highlighting. Thus we created a little tool which is able to derive such regex queries from Java and C# code as well as from UML-like interface descriptions. However, it is important to remember that even regular expression searches are still very limited compared to signature matching. In particular, since it is not possible to ignore the parameter order, signature- and interface-based searches are still not possible.

4.2.2 LIMITATIONS

It should have become clear in the recent subsections that mainstream search engines and most of the “first-generation” keyword-based search engines are not optimized for the kind of component retrieval we require for delivering components based on a (KobrA) specification. If this was the case, there would be no need to work on better retrieval solutions. Furthermore, as we discussed above, the mainstream search engines obviously have been trying to remove source code from their indices. Steele [Ste01] explains that the web is simply becoming too large to index all its content deeply enough, so there is a need for specialized or so-called “vertical” search engines. WebCVS systems, which require a large number of package hierarchy levels to be navigated to find code are a good example of this. Other limitations of general search engines like a user interface that is optimized on keywords and not on component descriptions or the retrieval and ranking algorithms (like Google's Pagerank [Pag98]) that are optimized for prose text make component retrieval too complicated for serious software development. Even with the tricks described above, the signature and interface matching capabilities with regular

search engines are limited. For example, it is not possible to match signatures in orders different to the one defined in a query. This is a serious limitation which can only be circumvented by adding all possible permutations for all parameter orders disjunctively, an approach that quickly takes queries beyond the maximum supported length.

Some other problems are not as obvious, but nevertheless make the use of these engines impractical to use for serious software engineering. For instance, Google and Yahoo both used to offer a Java-based API for automated access to their indices, but both were very unreliable and often delivered only cut-down versions of the actual result set. Apparently, neither is supported any more and have recently been replaced by other technologies. Moreover, since the filetypes of the programming languages are not officially supported there is no guarantee that they will be supported in the future nor that they will work reliably. Furthermore, some file extensions like “cs” are not limited to C# source files but also for other kinds of files and hence undesirable files might frequently be included in result sets. Additionally, it looks as if the filters only consider the file extension or even worse only the last characters of the URL for their decision about the type, which has the consequence that sometimes HTML files and other types slip through. These problems triggered the decision to build our own vertical search engine we describe in the next section.

4.3 THE BUILD-UP OF MEROBASE.COM

When we started the investigations of the web as a source for reusable software, we focussed on retrieving code using mainstream search engines. As we have shown in the previous part of this chapter, this approach works reasonably well for scientific purposes, but has significant drawbacks once a reliable tool is required. The main problem is that the two big players, Google and Yahoo, artificially limit the number of results they deliver for a search and additionally only offer a limited number of queries per day. This makes it impossible to use a tool is based on these two engines in a serious software production environment. Furthermore, the frequent changes of their content as discussed in section 4.1 also limited their usefulness for scientific comparisons. Hence, the collection of our own component base and the development of an own search engine was a natural next step.

4.3.1 CRAWLING AND INDEX STRUCTURE

With the recent advent of nutch and Lucene [Hat04], two powerful open source tools for the creation of search engines have become available. Although it would be feasible to store the crawl results in a relational database, Lucene offers some significant advantages. The most important benefit is that it offers a very fast full-text search capability, which is vital for all search engines as understood today (and for the implementation of information retrieval approaches for component searches). Since Lucene supports values to be stored in different fields it allows faceted, attribute value and even catalogue-based retrieval approaches (as we discussed in section 3.1.1) to be implemented. The drawback is that these fields are not relational as in a database and thus relational searches are not directly feasible. However, below we discuss how it is possible to overcome this problem to a certain extent with a special index structure. To solve the representation problem for the components in our index and to increase the realm of potential searches we combined all four representation methods described in the literature (and explained in section 3.1.1). The following table gives an overview of some of the most important fields

in our index and the data (or metadata) stored in them. It is important to mention that the content of all fields can be extracted automatically so that no human interaction is necessary.

Field	Representation Method	Content
content	free-text	source code
name	attribute value	component names
method	attribute value	method names
url	attribute value	component's URL
lang	faceted	component's programming language
kind	faceted	special kind of component, e.g. application or test case
methodSignature	attribute value	full signature of methods
namespace	enumerated	a component's namespace

Table 4.4: Exemplary fields contained in the Merobase index.

These fields are contained in each Lucene document, representing an individual component (which is typically a class). One specific field can be added several times to a document, i.e. a component can contain a number of methods with each of their names stored in a method field. In principle, on all of these fields the full Lucene query syntax (as e.g. described in [Hat04]) with wildcards, range queries etc. is applicable as long as the fields can be tokenized. However, one of Lucene's limitations becomes apparent at this point. Since no relational connections between fields are feasible, it is not possible to relate a parameter to a method signature. Interface-based retrieval, however, requires the ability to search for the exact signature of a method, including name, parameters and return type. Thus, we were forced to concatenate and store them in one field which is not tokenized to enable such exact matches. In turn, this means that only exact matches are possible with this structure and different parameter orders cannot be searched. However, by sorting the parameters alphabetically it is possible to also identify different orders of method parameters. We had to develop a number of further innovative solutions to fully implement all semantic search use cases which we will discuss in the next chapter. The associated extensions of our Lucene structure will also be discussed there.

The creation of the index using the nutch web crawler suite is straightforward. It includes powerful tools for traversing and managing links as well as for the interpretation of robots.txt files (which may restrict the access of search engines to websites) directly out of the box. However, as we pointed out earlier, we estimate that only about 1 of 10,000 documents on the web contains a source file interesting for us and thus a blind crawl would require far too much effort. Hence we fed the nutch engine with seed pages containing links to popular web service or component catalogues, for example. Another promising technique to find appropriate input is to “metasearch” general-style search engines as described above, which was also used by e.g. [Sea98]. Once an initial list of source files is found, it is useful to trim the URLs within them to find lists of further source files in higher directory levels. Luckily, the crawling of CVS and SVN repositories is simpler (once a functioning CVS and SVN client is available for integration into the crawler) and the access to one server often provides thousands of files in one pass (cf. table 4.6). Unfortunately, downloading files from CVS or SVN has a large overhead due to the lengthy login procedure required. For this reason we decided to cache the crawled content (which is about 120

GB) of these repositories locally in order to guarantee fast access to the source code. In principle, this would make sense for http-based files as well in order to avoid dead links, but we have not implemented this so far.

4.4 THE CONTENT OF MEROBASE

Given the large variety of assets that mainstream search engines are expected to index, it is clear that there is room for more specialized search services. Recently, this idea became popular under the name of so-called vertical search engines. While general or horizontal search engines cover a wide range of assets only shallowly, a vertical search engine is supposed to build a much deeper index on a smaller area of interest. Our software component search engine has grown to one of the largest source code and component collections available on the web. The following table gives an overview of its contents in summer 2007:

Programming Language	No. of Files	Percentage
Java	8,011,883	79.566%
<i>Source</i>	<i>3,927,475</i>	<i>49.021%</i>
<i>Binary</i>	<i>4,084,408</i>	<i>50.979%</i>
C#	207,092	2.057%
C	1,399,455	13.898%
WSDL	3,228	0.032%
.NET assemblies	447,801	4.447%
Total	10,069,459	100%

Table 4.5: Number of components/services indexed in Merobase in summer 2007.

Since we focused our initial crawling on Java, the numbers shown are certainly not representative of the distribution of files on the Internet. We not only indexed files available via HTTP, but also files stored in the CVS and Subversion (SVN) repositories of large open source hosters. While more than one million files (to be exact: 1,279,362) have been found on the open web via extensive crawling, by far the largest number of files has been retrieved from the various CVS servers as shown in the following table:

Hoster	CVS	SVN	total
java.net	3,159,151	0	3,159,151
sourceforge.net	2,193,030	208,083	2,401,113
apache.org	0	666,808	666,808
googlecode.com	0	348,584	348,584
eclipse.org	325,119	0	325,119
netbeans.org	31,275	0	31,275
tigris.org	13,159	9,711	22,870
savannah.nongnu.org	13,653	0	13,653

Hoster	CVS	SVN	total
savannah.gnu.org	9,425	0	9,425
gna.org	1,886	1,224	3,110

Table 4.6: Overview of components found in version control repositories.

Since we have focused our research efforts on the Java programming language we present some more detailed analyses of the distribution of files in this language below. As the goal was to deliver working (i.e. executable) components, another interesting number is the percentage of Java interfaces contained in our index. Java 5 also introduced the concept of a so-called enum(eration). However, this is clearly not very widely used so far since not a single enum is contained in our index as table 4.7 demonstrates below.

Type	Occurrences	Percentage
class	7,036,451	87.83%
enum	0	0.00%
interface	975,432	12.17%
total	8,011,883	100.00%

Table 4.7: Percentage of interfaces contained in all Java files.

The open source movement [Ray97] has certainly been the main trigger that has enabled the creation of large-scale component search engines. However, some open source licenses such as the General Public License (GPL) can become problematic for their users if they want to reuse material in proprietary projects since GPL-like licenses require the disclosure of new code that uses the original code. This can become dangerous for proprietary commercial projects that accidentally (or on purpose) used GPL'ed material. Consequently, it is a requirement for a code search engine to be able to search for a specific open search license or even better to exclude a group of licenses. To implement this requirement, we developed a regular expression-based recognition feature for open source licenses, which enabled us to construct the following statistics about the usage of open source licenses in our index. Since binary files do not contain this information, these numbers are based on the roughly 4 million Java source files in our index. In total we searched for 102 open source licenses gathered from various websites (such as opensource.org) and discovered components using 34 of them in our index. The following table contains the ten most popular licenses. Since the bulk of our indexed source files originates from designated open source hosting sites, one might assume that most of these files are annotated with an appropriate open source license. However, as the results of our analysis show, this is not the case as almost three quarters of all files do not contain a dedicated open source license.

License	Occurrence	Percentage
no license	2,880,932	73.35%
GNU General Public License	464,353	11.82%
Apache License, Version 2.0	209,878	5.34%
GNU Lesser General Public License	134,619	3.43%
Eclipse Public License v1.0	128,579	3.27%
Common Public License	42,147	1.07%

License	Occurrence	Percentage
BSD License	14,587	0.37%
Mozilla Public License Version 1.1	10,151	0.26%
Academic Free License	5,928	0.15%
Open Software License	5,919	0.15%
Sun Public License	4,756	0.12%
others	25,626	0.65%

Table 4.8: Open source licenses recognized for Java source files.

In order to execute and test components (for example, to support our Extreme Harvesting approach, see section 5.2) it is also interesting to know which classes can cause problems when executed remotely on a server – that is, when the Java sandbox would block file or network access and thus raise a SecurityException. We found that the following numbers of source files use classes from major Java IO packages:

Package	Occurrences
io	868,014
net	221,144
nio	18,102

Table 4.9: Number of classes using IO-Packages.

The usage of a graphical user interface (GUI) can cause similar problems as it requires access to a display and normally also user interaction, which is typically not available for remotely executed test cases. Thus, we investigated how many source classes can be executed without the necessity of requiring a display. The following table summarizes the usage of GUI frameworks found in our index:

Package	Occurrences
swing	417,475
awt	687,980
swt	69,306

Table 4.10: Overview of GUI frameworks used.

Since the SWT framework was introduced with Eclipse, we found about 32,000 source files from within one of the many Eclipse packages, but only about 37,000 classes outside the direct environment of Eclipse use the SWT. It is important to mention that these numbers are not independent since, for example, a component that uses Swing GUI elements typically also requires listener classes from the older AWT framework or, of course, can use file I/O as well.

Java has grown to a language with a large number of different target platforms (J2SE, J2EE, J2ME) and thus we were curious to find out how these are represented in our index. Unfortunately, it is not possible to determine directly which edition of Java a class is intended to be used with since all three share a number of libraries. However, we were able to recognize a number of special component types such as applets or applications (classes containing a main method) etc. as shown in the following table:

Type	Occurences	Percentage
Application	508,466	6.35%
Test Case	123,881	1.55%
Applet	91,224	1.14%
Servlet	18,311	0.23%
EJB	3,786	0.05%
MIDlet	1,660	0.02%
<i>All Java</i>	<i>8,011,883</i>	

Table 4.11: Distribution of special component types in the Merobase index.

452,295 source files contain a main method and thus can potentially be executed standalone in J2SE after compilation. In contrast, only 56,171 binary classes contain a main method. We speculate that this difference is due to the fact that most of our binaries classes originate from JAR libraries that typically only have one or a few entry points. As we pointed out earlier, we believe that a lot of files from our open web crawls are simple (teaching) examples that typically need to be executable and hence the ratio of files containing a main method is much higher here. The relatively large number of JUnit test cases indicates a high level of acceptance of this framework in the Java community. The small amount of Enterprise Java Beans (EJB) demonstrates again how difficult it still is to find reusable business components.

4.5 SHARING COMPONENTS OVER THE WEB

As shown in the previous part of this chapter, developers today have various opportunities to find reusable material on the web. However, it is still nearly impossible to publish reusable material in a targeted fashion for others. To be more precise, it is simple to publish material on the web, but hard to make it findable for others in a controlled manner. In principle, there are three ways in which components can be shared via the Internet:

1. Creating an open source project on Sourceforge or a similar site
2. Publishing components on a website and registering them with a search engine
3. Sharing files on a Peer-to-Peer network

However, all three possibilities are introduce uncertainties. New open source projects on Sourceforge (and similar sites) have to be manually approved, which induces a delay of perhaps a few days in the best case and a rejection in the worst case. Thus there is a high administrative overhead involved in making a project available online. Furthermore, the publication of a project on an open source hosting site does not necessarily mean that files will be quickly accessible through mainstream search engines. A project that we made publicly available in 2005 was, after months, indexed neither by Google nor by Yahoo. Only Google was able to retrieve the link to the project's homepage. After 6 months, Yahoo also delivered the homepage, but neither of them indexed the sources which are linked to directly from the project's homepage. As mentioned before, Google has obviously started to remove source files from its index. On the other hand, vertical search engines such as Koders and Krugle are known to update their indices only at very irregular intervals. As of 2007, our project was only findable via Krugle, who officially partnered with Sourceforge for code searches in 2006 (and thus should have privileged access to

the repository servers) and was not discoverable in Google Codesearch, Koders nor in Merobase. Thus, it is still highly unpredictable whether and when an uploaded project will be findable on the Internet.

We experimented with the second option by manually placing the aforementioned project on a .com and a university webpage and submitted the URLs to Google, Yahoo and MSN in November 2005 to check whether and when the search engines would be able to retrieve source code from the project. The results were also disappointing since it took about one month until at least the starting page and some of the linked source files became available. However, they were only sometimes reachable and not on a regular basis. This experience with mainstream search engines for non human readable files was recently acknowledged by the web service community where [Son07] reported similar behaviour for WSDL files. These observations make it clear that contributing components to the ubiquitous repository World Wide Web in a controlled fashion is not practical at present.

For the third option, we also investigated whether the common peer-to-peer (P2P) platform Gnutella is useful for component distribution, as P2P systems typically are a place where all kinds of files can easily be shared with almost no effort. Such peer-to-peer systems (P2P), which formerly started out with the famous Napster and were very successful in the late 1990s, are an appealing approach for solving the software repository problem. However, the results in 2005 were not encouraging. P2P systems like Gnutella (having almost 2 million users at that time according to [Men05]) are not suitable for source code searches at all. Although we were able to limit searches to a desired programming language with special search terms (like "class") and special filter settings (".java" or ".jar"), the results were not usable. Our investigations in December 2005 revealed only about 2,500 Java source files and about 1,100 JAR files on the Gnutella network. But, since P2P systems simply search in the name and not in the content of files they offer only the most simplistic search support and hence do not offer much incentive for developers to use P2P systems for this purpose.

The investigations in this subsection show that as of 2007 a well aimed sharing of open source components through search engines is still very difficult. Not to mention the attempt to make commercial binary components searchable over the Internet. Thus, there is no doubt that there is still plenty of room for a dedicated component storage solution that combines version control repositories with search for open source software and for a specialized, trusted brokerage solution as we will sketch it in section 9.3 with respect to commercial components.

5 SEMANTIC COMPONENT SEARCHING

*Don't mind Pierce and Hunnicutt, they're both first rate surgeons.
Sure, they'll show up to role call in their bathrobes.
They keep a still in their tent. Once they ran all my underwear up the flagpole.
But I want you to understand it's an honour to serve with these men.*
-- Major Margaret Houlihan, M.A.S.H.

In the last two chapters we discussed the state of the art in component retrieval and presented a potential solution to the repository problem. However, the explosion in the number of searchable components available on the Internet and in companies' repositories makes the retrieval problem even more pressing. In the past, when component repositories used to contain a few hundred elements, simple browsing-based retrieval techniques worked reasonably well, although they were certainly not perfect. But today's repositories which are more than a thousand times larger impose new challenges on retrieval techniques in terms of precision. Let us illustrate this by means of signature matching [Zar95], a well-known and understood retrieval technique from the 1990s. When we apply it to our Merobase index and search for a `Stack` component with methods for pushing and popping integers, we receive more than 40,000 results of which only about one hundred are likely to deliver the required functionality. Thus, browsing through the results to find components that are actually `Stacks` is similar to finding a needle in a haystack.

Furthermore, with the experience, we gathered during the development and operation of Merobase we realized that previous component retrieval approaches were somewhat unspecific about the use cases which they expected the system to support. In other words, they had no idea how people would use such a retrieval system in practice since none of these systems ever made it into practical use. This, in turn, makes it very difficult to optimize a search engine towards semantic component retrieval, which we defined earlier in this thesis as the task of delivering candidate components that best fulfil the purpose a developer has in mind. Through the development of Merobase we were able to observe user behaviour and, combining this with our own experience from using the system, we were able to derive a number of requirements that today's large-scale component search engines should support. We introduce them in the next section before explaining how it is possible to define search algorithms that they deserve to be regarded as semantic component retrieval approaches.

5.1 USE CASES FOR COMPONENT SEARCH ENGINES

Depending on the point of time in the development process at which a search is performed, more or less information about the desired component is known. Early in the process, when perhaps just a very coarse assignment of responsibilities has been performed, a component search engine is more likely to be used to provide the user with an impression of what of components are around in the repository. Neither a detailed syntactical description nor a description of the semantics is likely to be available for the component under discussion. The other extreme occurs much later in the development process after the design for a component under discussion has been finished. At this point, the developer is likely to have a clear mental picture and ideally a complete specification of the component desired. Thus, in this scenario a search engine needs to deliver very precise results, which should be usable without too much adaptation effort. However, after implementing our Merobase search engine, we realized that developers could have a third and even a fourth reason for using a component search engine. The third use case occurs when a developer wishes to find the source code of a very concrete class from some open source system, e.g. he/she might want to comprehend the internal flow in the component before using it. A search engine which is able to deliver the required files quickly is likely to save a lot of effort that otherwise would have to be invested into locating and downloading the appropriate source package and finding the appropriate file. The fourth use case we identified in the context of component search engines is similar to the third one and deals with finding the library that contains a specific class. In the context of Java, such libraries are often JAR files. The following table summarizes these different usage modes. We explain each of them in more detail later where necessary.

Use Case	Description	Useful for...
1) speculative searches	the search engine is used to get an impression of what is available and what might be a good design for a component	Design & coding
2) definitive searches	a component fulfilling a given specification is wanted	Coding and testing
3) concrete open source searches	the source code of a specific class from a concrete open source system is required e.g. to better understand how it is used	Coding
4) library searches	the user looks for the library containing a specific class, e.g. triggered by a <code>ClassNotFoundException</code>	Coding, testing/deployment

Table 5.1: Use cases for component search engines.

For the sake of completeness we should briefly mention textual searches at this point. although they can neither be called semantic in the sense of this thesis and no longer present any serious challenges. There exist open source search frameworks such as Lucene [Hat04] and others which specialized on this kind of search and provide impressive performance. For example, Lucene can typically search an index of millions of documents in less than five seconds. Since it has developed into a full-grown text search engine over the last years and is optimized for this task, Lucene is the optimal choice for supporting textual searches. By “textual search” we mean searches for a specific string of text within the subject

artefacts. Such a query might be useful in our context to find out how a specific class is used, for example. Like most search engines, Lucene allows a search to be constrained to specific fields by placing a string in double quotes. Thus, the query “new `BufferedReader`” would, for example, deliver source code that instantiates a `BufferedReader`. Adding the `lang:java` constraint would only return results in the Java programming language. By default, Lucene searches in the content field, although it is always possible to define other fields that should be searched as just shown with the constraint. This search mode is extremely simple to implement since the expected result can be found one for one within the source code. Furthermore, Lucene even offers support for wildcards and a number of other interesting features. Details can be found in [Hat04]. The other use cases are much more interesting from a scientific point of view since they require special extensions of Lucene's concepts and thus will be described in more detail in the following.

5.1.1 SPECULATIVE AND OPEN SOURCE SEARCHES

At first sight, speculative and open source searches seem rather contrary. However, since the same heuristics are applicable in both cases to optimize the search algorithm, we will discuss them together in this section. Before going into more detail about the implementation, we have to distinguish between speculative searches and textual searches. We realized that the simple keyword-matching of Lucene's algorithm introduced in the subsection before is typically not sufficient to provide meaningful results for developers. Since speculative searches are typically used to get an impression of what is available in a repository, a developer who searches for `Stack` is likely to be interested in an implementation of a `Stack` and not in components where `Stack` appears somewhere in the source code because it is, for example, used there etc. This is similar to the optimizations of modern web search engines, especially Google, to deliver “meaningful” results. The Google website¹⁰ states that “*Google tries to find pages that are both reputable and relevant*”. Reputable sites are delivered from the well-known Pagerank algorithm which was published in [Pag98] when Google was more of an academic research project than a commercially oriented company. However, Pagerank is one reason for Google's success. Finding the most relevant results for a query is at least as important as ranking the most reputable sites first. Google does this by paying particular attention to some special elements of web pages, such as headlines or the title of pages. We have found that a similar idea is applicable to software components as well. The key in doing so successfully is to identify the elements that deserve the accentuation.

Consider the following simple example for a better understanding. When someone types the query “`eclipse astparser`”, he or she is likely to be interested in that specific class from the Eclipse project. Hence, the indexed versions of the `ASTParser` class should be returned first, simply due to their relevance for this query. However, when we started to assess the performance of our early Merobase prototypes, we quickly realized that a simple keyword-matching approach is not enough to deliver semantically relevant information for such a query. Similarly, suppose the user types “`stack`” as a speculative query. What is he or she most likely to be looking for? We believe it is components delivering the LIFO functionality of the well known `Stack` abstract data type. However, purely text-based searches deliver all results that contain the text “`stack`” somewhere in the source code, even if they have not even the slightest resemblance to this abstraction.

¹⁰ http://www.google.com/librariancenter/articles/0512_01.html

However, since these two main usage scenarios are not supported by the standard Lucene approach, an important contribution of this thesis is development of query algorithms to support these use cases. As they have similar requirements they can both be supported by the same basic techniques. The basic idea is to transfer the above mentioned techniques, applied in mainstream search engines, to our component search engine. Thus, we experimented with expanding our queries to selected metadata fields in our index and assigned them different weights to emphasize their relative importance. The general formula for the relevance R of a document under consideration is as follows:

$$R = \sum_{n=1}^k w_n \cdot r_n$$

Whereby r_n is the relevance for each field as delivered by Lucene, w_n is the weight assigned to each field and k is the number of fields used. This, of course, can be normalized to a value between 0 and 1 by:

$$R = \frac{\sum_{n=1}^k w_n \cdot r_n}{k \cdot \sum_{n=1}^k w_n}$$

In our early prototypes we had to write a specific search routine to perform this weighting, but in more recent versions Lucene added a special `QueryParser` making it possible to search over more than one field and to attach different weights to them. We experimented with various values and found that the following configuration works equally well for both speculative and open source searches:

n	Field	Weight w_n
1	name	5
2	namespace	3
3	interface	4
4	project	4
5	method	1
6	content	0.5
7	url	4

Table 5.2: Fields and weights used for improved ranking within speculative and open source searches.

The general rationale behind the choice for the fields and the associated weight values shown is that classes in object-oriented programming are supposed to be abstractions of domain objects. However, depending on the individual developer and the task he or she has to deal with, functionality can be implemented on various levels within a system and thus it also makes sense to assign high weights to the namespace or the whole project. The interface field indicates the name of implemented interfaces and thus is also a good source of information as well as the project name. However, sometimes simple functionality is implemented in just one method. The content itself should not be totally left out since some helpful terms might be hidden in the source or in a comment. The URL is also helpful to ensure

that relevant open source classes are ranked highly since the namespace and the name are both normally included in the URL. As our evaluation in chapter 7 demonstrates, the shown combination of weights works well for both use cases.

5.1.2 DEFINITIVE SEARCHES

Definitive searches are the counterpart to speculative searches and are typically used when a concrete specification for a component has already been defined, typically as part of the overall design of a system (e.g. as recommended by [Atk02]). Since definitive (or specification-based) queries require the search engine to “understand” the syntax – and in the ideal case also the semantics – of programming languages they are more expensive to implement than simple keyword-based algorithms. To be able to understand the syntactical structure of a program, typically a parser is required for each supported language in order to extract the required information during index creation. Assessing whether a component is semantically appropriate for the purpose at hand is even more complicated and at the time of writing has not been supported by any publicly available component search engine. Since the semantic matching of components is the major contribution of this thesis, we leave this aspect aside at this point and discuss it in more detail in section 5.2 et seq.

The next challenge to support syntactical definitive searches, is to find an appropriate representation format: a simple approximation for full syntactic searches would of course be a name-based search (as e.g. offered by Krugle and Koders) where only class and method names are saved in appropriate fields that can easily be stored in databases as well as in an Lucene index. Storing the complete interface information is more complicated, however. It either requires a relational database schema with quite a number of join operations for searches or some heuristics to store this in Lucene appropriately. Since Lucene is required for keyword-matching, a combination of both approaches seems to be the most powerful option. However, we believe this is not practically implementable for a variety of reasons. The biggest issues we see in the combination of both approaches is the overhead involved in storing the indexed information twice and the problems in merging results from two different searches together and ranking them. Thus, in the context of this thesis we implemented the pure Lucene version as described in the following.

The overall structure of our index has already been described in subsection 4.3.1 and basically comprises a field called *content*, storing the textual content (i.e. normally the source code) of a component, and a number of other fields containing metadata about the component. While most fields such as the class name or method names are rather simple, storing a method signature is not that simple with Lucene as it lacks the ability to store relationships between the fields. Our solution is to concatenate method names, parameter types and return values into a single field as shown in the following example. Supposed a component has a method *random* accepting two parameters of type *int* and returning one *int*. This would be mapped to the following entry in the method field

```
mn:random_rt:int_pt:int_pt:int
```

This field must be indexed directly (i.e. not tokenized as explained in 4.3.1) because otherwise Lucene would not be able to recognize it correctly. Furthermore, the parameter types have to be sorted

alphabetically since we want to recognize permuted parameter orders as well. It is obvious that search requests in Java or the UML-like query language have to be translated into this format by an appropriate parser before searches can be carried out. Unfortunately, wildcard searches on these structures are not possible due to internal restrictions of Lucene. However, it is possible to enable at least a pure signature matching [Zar95] for the operations by storing a second, largely identical field that does not contain the method name, but only the parameter and return types. In the case of the random method, this would have the form:

```
rt:int_pt:int_pt:int
```

Another problem that could occur in this context is when a given signature is required more than once in one class since it is not possible to specify the number of required appearances for Lucene. However, it is feasible to circumvent the problem by preceding each signature with a counter of the number of times it appears in the component, i.e.

```
1_rt:int_pt:int_pt:int  
2_rt:int_pt:int_pt:int
```

This makes it possible to search for classes that contain the required signature once as well as twice, or any other number of times. However, as soon as parameters or return values contain object types we again face the problem of identifier choices. Thus, for example, a `LifoBuffer` could be regarded as equivalent to a `Stack` on a purely textual basis (and in Lucene queries) if it appears as a parameter or return type. The only way to mitigate this problem is to replace the name of the current class with “this” if it occurs as a parameter or return type. Let us illustrate this problem with a little example. Suppose we are searching for a class `Matrix` containing a method with the following signature –

```
multiply(Matrix):Matrix
```

which would be mapped to –

```
rt:matrix_pt:matrix
```

in our Lucene representation. Unfortunately, based on this structure, Lucene would not be able to match this to a class with a method with a signature that is identical except for the class name (e.g. `MyMatrixImplementation` rather than `Matrix`). However, if all types that are identical to the class name itself are replaced by “this”, i.e. as

```
rt:this_pt:this
```

Lucene could at least recognize this as a potential candidate and the search engine tool would be able to test it for semantic equivalence as shown e.g. in table 7.3 in section 7.3. These workarounds are especially useful when implementing so-called multilevel searches, which we will describe in the following. It is intuitively clear that the more complex a component design becomes the less components are likely to match it completely [Sam97] and thus the recall decreases quickly with the size of component. Thus, we have developed some heuristics that relax the search criteria in multiple steps if an

insufficient number of components is found by the basic matching algorithm. The following table presents some examples of reasonable “relaxed search” heuristics. The dollar sign used in some places indicates either an OR separator or a wildcard.

Level	Approach	Example
0)	Original syntax	<pre>ShoppingCart(addItem(Item, int):void; total():double;)</pre>
1)	Split class name	<pre>ShoppingCart -> ShoppingCart\$Shopping\$Cart</pre>
2)	Class name is merely desired	<pre>ShoppingCart -> ShoppingCart\$</pre>
3)	Ignore method names	<pre>addItem(Item,int):void; -> \$(Item,int):void;</pre>

Table 5.3: Multi-level searching for specification-based searches.

Various other combinations of relaxed searches are also imaginable. However, the strategy shown above should be sufficient for a large index since the number of results normally increases quickly with each level. Even if this is not the case, it normally does not make sense to add too many relaxation levels since each level requires a completely new search, which usually requires around 3 seconds time. Fortunately, Lucene ranks components fulfilling more search criteria higher in the result list so that results that are likely to be closer to the original request are presented automatically first.

However, this approach still suffers from a number of limitations. The first one is obviously that searches are restricted to the syntactical information in a component's interface, which may contain misleading linguistic information in class and method names as well as in parameter and return type names. This introduces all the usual problems known from information retrieval [Fur87] such as the recognition of synonyms (i.e. different words with the same meaning), homonyms (same words with different meaning) or even hypernyms (one term is a generic term for another). The information retrieval community has tried to tackle this problem in two main ways. First, it has created large dictionaries, such as WordNet [Mil90], which make it possible to look up synonym and homonym information. Second, they have developed techniques like Latent Semantic Analysis (LSA) that analyse the complete content of the searchable artefacts [Dee90]. Since these are applicable for speculative searches as well as definitive searches, we discuss them in a separate subsection (cf. 5.1.4).

5.1.3 JAVA LIBRARY SEARCHES

Since Java does not disclose the required interface of classes or JAR files, it is typical for Java execution environments to create so-called `ClassNotFoundException`s when binary components are to be integrated into a system or appropriate error messages when a source code is to be compiled. Usually, the only information a developer gets in this context is the name of the missing class and the namespace, i.e. the package, it belongs to. Developers used to have to waste a lot of time guessing, browsing and searching to find the appropriate JAR file containing the required class. Since our search engine has indexed more than 4 million binary Java files from a large number of JAR files, we are able to find the

library containing a specific class by proving some additional constraints on a search. For example, a search for the class `org.apache.lucene.search.Query` has the following form on Merobase:

```
namespace:org.apache.lucene.search name:Query
```

Thus, a simple field-based, keyword-matching approach can be used for this task. Depending on whether the binary or the source version of the class should be retrieved, the query can either be extended by the constraint *form:binary* or by the constraint *form:source*. The default case delivers both variants. Table 8.2 shows a comparison of our data content with some other public search engines regarding their support for this specialized task later.

5.1.4 FURTHER OPTIMIZATION OPTIONS

Intuitively, it is clear that the more complex the specification of a component becomes the smaller the amount of delivered reuse candidates gets [Sam97]. Since similar problems are well-known in the IR community, a number of solutions are already present there which we have also adopted to increase the recall of our system. A simple technique to increase the recall of keyword-based searches is to search for synonyms of the desired keywords as well. This is subsumed under various techniques for query expansion as discussed in [Bae99]. While there are statistical ways of automatically creating thesauri of terms via co-occurrence matrices, the process to create such a matrix is very computation-intensive and thus currently only an interesting option for future research. Latent Semantic Indexing (LSI) is another well-known information retrieval technique developed by [Dee90], which is based on the co-occurrence of terms and is supposed to mitigate these problems. It is based on correlation analysis of term document vectors and thus is expected to extract concepts out of documents rather than just plain terms. Thus, it is able to create an “understanding” of what the documents are about and synonyms, for instance, are automatically understood correctly. In the context of this dissertation, we have experimented with this approach for our component searches (details can be found in [Gru06]). However, our results seem to confirm the results of [Ye01] who found that LSI only functions reasonably in the component retrieval context if additional text (i.e. in Ye's case comments) is available. In other words, LSI usually works better if developers enter small stories, describing the functionality they need, rather than just a class name. However, in our trials [Gru07] it was so hard to achieve acceptable precision, due to the large amount of noise usually delivered, that we did not investigate LSI further.

Manually collected thesauri for English, such as WordNet [Mil90], are also freely available on the web and we have also experimented with their use for query expansion in the context of component searching. However, our experiments have shown that this is also not very promising. First of all, programmers often tend to use descriptive names for classes and operations which are assembled from various terms which are typically not contained in a dictionary as a whole. The Java coding guidelines promote the use of the so-called “camel case” identifiers (e.g. `isCourseToBeScheduled`) for this purpose, i.e. an upper case letter should be used for each new word that is attached to another one. As we have already discussed, it is indeed possible to decompose such constructs and use them for relaxed searches if the composite term could not be found. However, the decomposition of the camel case typically yields a number of new terms to be searched for. However, these do not necessarily describe the original concept any more and might add a lot of noise to the search results, even if

“stopwords” [Bae99], such as “is” and “be” in the above example, were filtered out. Another problem that limits the applicability of thesauri for component searches is the fact that they often contain a large number of terms related to a concept that are usually not very helpful for discovering similar components. Consider, for example, a query for the term *stack* in WordNet that delivers the following list of synonyms:

- batch, deal, flock, good deal, great deal, hatful, heap, lot, mass, mess, mickle, mint, mountain, muckle, passel, peck, pile, plenty, pot, quite a little, raft, sight, slew, spate, stack, tidy sum, wad
- push-down list, push-down stack, stack
- smokestack, stack
- push-down storage, push-down store, stack

Although this list includes a separation of contexts for the synonyms, it is impossible to tell without human intervention which of the four categories is the most useful for a given context. Furthermore, even if this were possible, a potential number of twenty or more synonyms would still not be very helpful for directed component searches. Thus, we decided not to pursue this approach further in this dissertation and only included a manually collected list of a few dozen synonyms which often appears in the context of programming.

5.1.5 DESIGN RECOMMENDATIONS BASED ON SEARCH RESULTS

In recent years, online shops have popularized so-called collaborative filtering systems that recommend potentially interesting items to customers (“*people who bought that book also were interested in ...*”). Such systems typically require a list of users that have purchased a number of items and from this data pool recommendations can be derived for similar users. In general, there are two feasible approaches to implement such an algorithm [Bae99], namely user-based algorithms that cluster similar users together and item-based algorithms that maintain a matrix with item-item pairs containing information about how often two users bought these two items.

These techniques have not only been applied to shopping sites on the web, but also to software reuse problems already. [McC07] uses such an algorithm to recommend Java Swing method invocations based on the analysis of a few thousand Java classes. A class is regarded as a user by their system and the method invocations are the items he is interested in. However, at the time of writing it is not clear whether that approach is generalizable to multiple domains and to larger data collections. Hence we have used a slightly different approach in our search engine. We have found it feasible to extract the first n search results for a query and to derive an interface recommendation based on this information. The idea is to utilize the method signatures of these results to calculate the optimal average class interface in that context. We do this by counting how often a given method signature appears amongst the top n results and return the most popular signatures or all signatures that appear more often than a given threshold as the average of the n classes. A simple algorithm that delivers usable results is described by the following piece of pseudo code:

```
for the first n results
  for each signature of the current result
    if signature is stored in hashtable of signatures
      increase number of occurrences
    else
      store signature in hashtable
  next signature
next result
for each signature in the hashtable
  if number of occurrences > threshold
    add signature to result
  endif
next signature
```

This algorithm works reasonable fast (i.e. typically a response time of less than a second is required since the recommendation has to be created on the fly when a search is run) and well for $n = 100$ and a threshold of appearances of 20% of the components. Consider, as an example, a speculative search for a “stack” component for which this algorithm recommends the following interface:

```
public class Stack{
    boolean isEmpty() {}
    Object pop() {}
    void push(Object arg1) {}
    Object top() {}
}
```

It should be obvious that common abstractions such as `Stack` deliver better results than a search for e.g. “public” would, but that is in the nature of things. Unfortunately, in its current implementation the algorithm is merely able to recognize identical method signatures and thus is rather limited. Although we have implemented a more sophisticated analysis of signatures and method names, it is too time consuming in the context of a component search engine since it can require more than 10 seconds. This time might be acceptable for transparent proactive searches that are triggered in an IDE without knowledge of the user or for future research, but at present it is certainly too long for a web-based search engine.

5.2 SPECIFICATION-BASED RETRIEVAL WITH EXTREME HARVESTING

Probably the most important contribution of this thesis is the development of a specification-based reuse approach that is convincing in its ease of use as well and uses the vast amount of (open source) software on the World Wide Web as a component repository. Nevertheless, our approach is neither limited to open source software nor reliant upon the web, i.e. it can also be used in proprietary CVS repositories, for instance. Although the syntactical definitive searches, which we introduced in section 5.1.2, come much closer to this goal than previous retrieval techniques as we will see in the evaluation of our work in

chapter 7 their precision is generally still below 50 percent or less and the recall can also decrease rapidly for more complex queries. Thus, it is desirable to further extend the search techniques introduced so far.

The literature identifies only two approaches capable of guaranteeing a precision of 100 percent or at least close to this optimum, namely formal semantics based (cf. e.g. [Zar97] or [Fis91]) and operational approaches. The main problem with the former is that it is difficult to apply and that the semantics of all components have to be specified manually, a task that is certainly too expensive for large numbers of components. Most approaches of the latter category are based on behaviour sampling [Pod93] where random samples of the input space are used to identify functions to deliver the expected results. However, while this was doable for small test collections of about 100 C functions, it is certainly not feasible for millions of components in a practical amount of time. Thus, for this thesis we extended our syntax-driven retrieval techniques from the previous part of this chapter with ideas inspired by behaviour sampling. We have found that it is possible to use the fast and relatively cheap interface-driven retrieval approach as a filtering process before using an expensive operational retrieval approach. Moreover, we have discovered that Extreme Programming (XP, [Bec99]) offers the optimal context (with its maxim “*design a little, test a little, code a little*”) for such a search approach since it creates an operational semantic description of all units under development. With a slight change, we can adapt XP’s guiding principle to become “design a little, test a little, *reuse* a little” and our approach can be integrated easily into any test-driven development process and even transparently into development environments and tools. Due to its natural relationship to Extreme Programming we have called our approach Extreme Harvesting.

5.2.1 PROCESS OVERVIEW

The figure below provides a schematic summary of the main steps involved in the practical implementation of our approach as initially introduced in [Hum04]. Only the steps (a) and (b) need to be performed manually as part of the software design process. Steps (c) to (f) can be processed automatically by a tool.

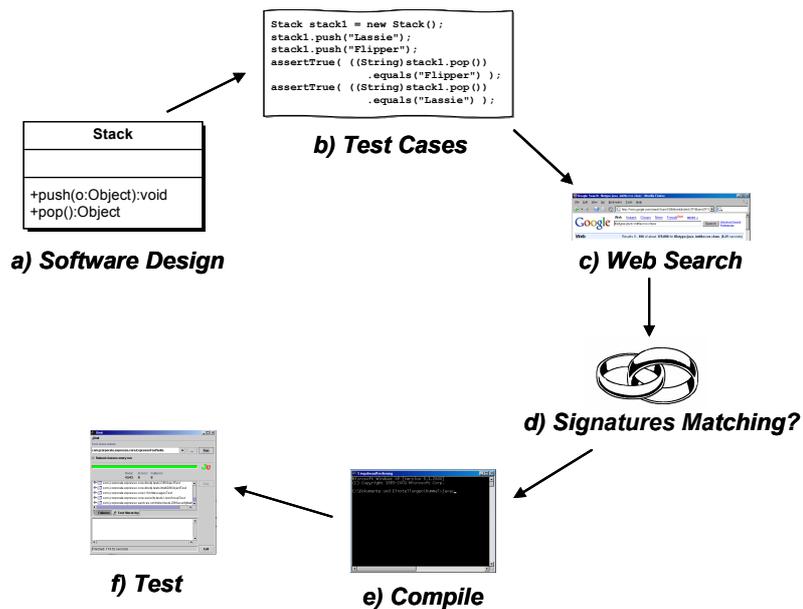


Figure 5.1: Schematic description of Extreme Harvesting process.

The steps shown in the figure have the following meaning:

- define syntactic signature of desired component
- define semantics of desired component in terms of test cases
- search for candidate components using an arbitrary search engine with a search term derived from (a)
- find source units which have the exact signature defined in (a)
- filter out components which are not valid (i.e. not compilable) source units, if necessary, try to find any other units upon which the matching component relies for execution
- establish which components are semantically acceptable (or closest to the requirements) by applying the tests defined in (b)

It is important to note that the search step (c) does not necessarily need to be carried out with the help of Google or another general-style web search engine. A specialized engine or a proprietary repository can be plugged in at this point as well. In fact, the use of our Merobase repository with its optimized capabilities for interface-driven searches makes step (d) widely superfluous in the best case. Another recent development in the context of this dissertation [Kru07] makes it even possible to renounce step (a) and to automatically extract the interface of the unit under test from the test case and thus deserves the label *test-driven reuse*. We will provide a more detailed explanation of the steps in the harvesting process below.

5.2.2 COMPONENT SEARCHING – A HYBRID APPROACH

Based on the lessons learned from previous approaches in section 3.2, we have created Extreme Harvesting as a new hybrid semantics-driven search and retrieval approach by integrating some of the

techniques outlined there. As stated in [Mil98], a retrieval process typically involves two criteria as a candidate component may fulfil the *matching condition* of one specific retrieval technique, but may not necessarily match a user's *relevance criterion* (recall the “conceptual gap problem”). For example, an information retrieval-based technique might retrieve 20 components matching the term “customer” but only two of them might actually fulfil the user's requirements and thereby his relevance criterion. In other words, a single matching criterion is typically too weak to guarantee satisfactory precision. Hence, applying more than one matching criterion can essentially be understood as a filtering process that iteratively reduces the number of components in the result set until only acceptable components are left. In the first version of the harvesting tool, we applied three filtering stages, namely linguistic, syntactic and semantic filtering, as shown in the sketch below:

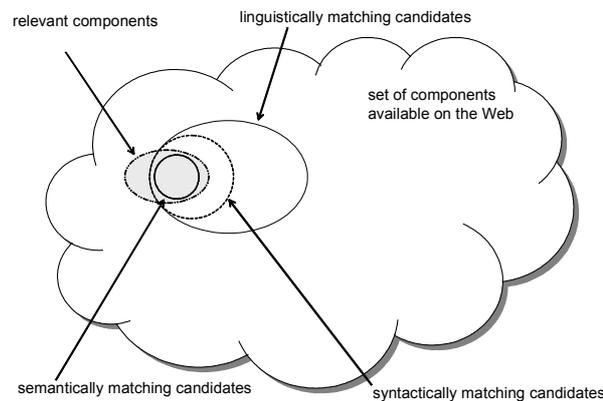


Figure 5.2: Search space reduction.

The cost of applying these filtering steps grows in the order they are introduced. For this reason the combination of the three steps is the only practical way to retrieve components semantically from the web or a large-scale software repository (without dedicated support for interface-based searches). In theory, other combinations of retrieval techniques are also imaginable, but ultimately it is always a trade-off between the effort at index creation time and the time required to respond to a query. Since we were able to store the interfaces of components in a later version of our Merobase index, we are able to omit one filtering step and retrieve tested results more quickly since. It would also be conceivable to even perform the compilation of components at crawl time and store dependencies and binary files to reduce the testing time even further, but this would require an amount of effort at crawl time that has not been viable so far.

5.3 COMPONENT EVALUATION

For general search engines it is normally sufficient if they deliver results that are reasonably close to what the user expects. However, component search engines are usually expected to deliver a much higher precision since even if only small adaptations to component are necessary to reuse them, this can become very expensive. Thus various heuristics are discussed in section 4.3 that are intended to deliver results much closer to a specification than plain keyword-based searches (which is confirmed by our evaluation in chapter 7). However, the Extreme Harvesting approach, as we have just explained it, requires even

more sophisticated techniques to determine the “distance” between a query and a candidate component and ultimately also the possibility of adapting the candidate for execution. In other words, to be usable for the purpose at hand a component must ultimately fulfil a given specification and the software reuse community has been working on a measure of the degree of conformance of two components (or a query and a component) for many years.

This, however, is not as simple as it may sound. As we have pointed out, a component has various dimensions that all influence the degree to which a component matches the desired specification, but ultimately only its functional semantics are an indicator of whether or not it is fit for purpose. For example, the signatures or names in the component can differ from the specification even if the behaviour of the component finally fits, which makes it difficult to define a metric for the conformance of a component, even though there have been various attempts to solve at least one or the other aspect of this problem. For example, in her seminal work about subtyping relations [Lis93], Liskov defined a fundamental understanding of conformance in object-oriented inheritance relationships – that is, under what circumstances a superclass can be replaced by a subclass. However, this approach requires a knowledge of the complete inheritance hierarchy which is rarely available in large and not organized component collections. [Zar95] discussed signature matching for functional languages under the premise of reuse and of course the information retrieval community has been researching the closeness of textual documents to one another. However, a comprehensive approach for software component reuse is still not available. Thus, in the next subsections we will explain our approach for putting the available pieces together in order to develop a component matching and adoption mechanism applicable for Extreme Harvesting.

5.3.1 LINGUISTIC CONFORMANCE

Although a recent study has confirmed [Ami04] that about 85% of the class names of an open source project can be traced back to entries in the popular WordNet¹¹ dictionary [Mil90], the so-called “vocabulary problem” reported by Landauer et al. in [Lan87] still has to be taken into account. Simply stated, the authors discuss the fact that different programmers tend to use different names for their components. They report the probability of two people using the same term for a concept as being only about 20%. [Kra03] recently conducted a similar study in the context of method signatures and found the similarity of signatures and names based on the required functionality of around 40%. In a conventional (older) component repository with a relatively small number of assets this led to a serious reduction in the effectiveness of information retrieval methods, since most of the components are stored under just one name. An obvious way to tackle this is to use synonyms as we already discussed in section 5.1.4. By storing a component with an additional alias (i.e. synonym) the recall can be doubled, in theory. However, as Furnas et al. [Fur87] state “*many aliases are needed to achieve a really good performance*”, and this would, in turn, drastically decrease the precision. In addition, it is still not clear how usable aliases can be discovered.

However, we expect a given piece of functionality to appear with many different interfaces and names on the Internet since diversity is one of its major strong points and the potential number of different

¹¹ <http://wordnet.princeton.edu/>

implementations is typically large - at least for relatively simple components. Ultimately, however, this only shifts the problem to more complex components and thus it is still necessary to find good heuristics that are able to suggest components that have the closest possible match to the query if no direct matches are available. We have already established some simple ideas in the context of the multilevel search approach in section 5.1.2. However, the linguistic dimension is still worthy of further investigation. As we shall see later, a signature-driven approach to candidate selection which completely ignores methods and parameter names is able to increase the recall significantly, although it is still too expensive for practical use at the time of writing.

5.3.2 SIGNATURE MATCHES IN JAVA

Signature matching in its original form, as defined by [Zar95] for functional languages with firm type hierarchies, recognizes between two functions only when they were identical in terms of the types they used in their interfaces. To our knowledge, only [Str94] has transferred these ideas to an object-oriented language, namely Ada. We are not aware of any work in this direction for today's common languages such as Java or C#. However, it is rather straightforward to transfer the ideas from these older publications to Java, for instance. In general, for a Java class it is merely necessary to take the signatures of all its non-private methods into account to apply signature matching in a simple form. Unfortunately, potential breaches of the information hiding principle through non-private attributes in classes pose additional challenges to this approach. However, in our experience this issue occurs so rarely that we have decided to disregard it for our current prototype. For the example shown in figure 5.3, signature matching would work well to determine the counterparts of the get (signature: `int x int -> double`) and the set methods (signature: `int x int x double -> void`), but it would fail to choose the correct method for the add, sub and mul methods.

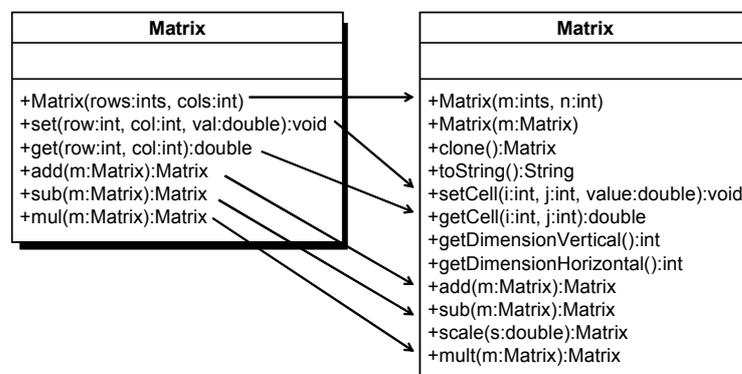


Figure 5.3: Example for a candidate result that requires an adapter.

The idea of relaxed signature matches that, for instance, also accept different parameter orders (as we already used in the creation of the Lucene index) is also transferable to modern object-oriented languages. Even the idea of relaxing parameter or return types is easily applicable for primitive types in Java as we will show below. In principle, this is also imaginable for object types based on subtyping relations as investigated by [Lis93], however, in practice it is difficult to fully establish the required inheritance hierarchies as we will discuss soon. The generally applicable rule in this context is that preconditions cannot be strengthened and postconditions cannot not be weakened in a subtype.

Translated to parameters in an operation signature this means that the parameter in a candidate can be “extended”, as shown in this example where for the query –

```
boolean isNegative(int number)
```

a candidate with the signature

```
boolean isNegative(long number)
```

would also be acceptable. The inverse principle can be applied for return values. If `long` was expected `int` would also be a feasible option. The following table shows the possibilities for the relaxation of primitive types in Java.

Relaxed Matches for Parameters		Relaxed Matches for Return Values	
Expected	Also Acceptable	Expected	Also Acceptable
char	String	String	char
byte	short, int, long	long	byte, short, int
short	int, long	int	byte, short
int	long	short	byte
float	double	double	float
byte, short, int	float	float	byte, short, int
byte, short, int, long	double	double	byte, short, int, long
Object	all object types		

Table 5.4: Possibilities for relaxed signature matches in Java.

The problem with object types in this context, however, lies in the difficulty of recognizing inheritance hierarchies in unstructured repositories and definitively finding the appropriate superclass. Consider a `Stack` class that inherits from a `Vector` for example. In theory, the `Stack` could thus replace the `Vector` as a parameter. However, in practice, only in very rare cases is the `Vector` class referenced in a fully qualified manner and would thus be precisely identifiable. Even if that were the case, there could still exist a large number of different versions or variants of that specific `Vector` in a repository that could have different properties. This is also a problem for signature matching in general, since, if an object type is used in a signature, it is not normally fully qualified nor are the signatures of its methods fully defined and available (cf. section 5.6). This is a reason why (unknown) object types can only be treated as textual elements in a signature matching approach with all the attendant linguistic problems discussed above.

Java 5 (also known as Java 1.5) has brought at least some relief in this context as it introduced a new feature called “autoboxing” which frees the developers from caring about the conversion of primitive types into the so-called wrapper types and vice versa. Another example should make clear what this is supposed to mean. Before Java 5 the following code was required:

```
Integer wrapper = new Integer(42);
int primitive = wrapper.intValue();
```

Since the introduction of autoboxing and the reverse auto-unboxing in 1.5 Java the following two statements are now feasible:

```
Integer wrapper = 42;
int primitive = wrapper;
```

In conclusion, signature matching for Java, as discussed above is another helpful technique to decrease the number of potential operation matches. However, it is still not sufficient on its own due to the peculiarities of object-oriented languages and their inability to represent the semantics of an operation. Thus, it is practically impossible to determine whether two operations match by purely comparing their signatures and the only viable option is to try all feasible permutations.

5.4 RESULT ADAPTATION

The adaptation of components has long been recognized as an important issue in software reuse, whether it be in the reuse community [Mil02], as a design pattern [GoF95], or even as the topic of PhD theses (e.g. [Gsc02]) or research papers (cf. [Mez01]). We can only briefly discuss the most important issues in the context of this dissertation and refer the reader to the works just mentioned for more detailed information. In our context a typical adaptation scenario occurs when a client wants to reuse an existing component whose interface does not exactly match his/her requirements. This means that an adapter is required as “glue code” between the client and the existing component. From the client’s point of view the adapter is supposed to offer exactly the interface he/she expects. The existing component, on the other hand, expects the adapter to use it in exactly the way it was designed to be used. When this is the case, both the client and reused component can be “satisfied” without changing any of their code. The goal of Extreme Harvesting is to fully automate this process of adapter creation so that a user does not have to be concerned with building it at all.

5.4.1 GoF ADAPTERS

Probably the most well-known discussion on adapters can be found in the famous Gang of Four (GoF) design pattern book [GoF95] where two forms of the so-called adapter pattern are mentioned. The more intuitive implementation of the two is probably the object adapter which is shown in figure 5.4.

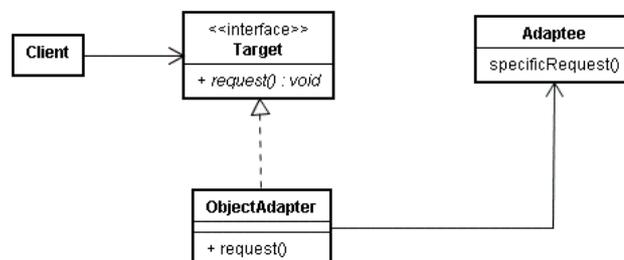


Figure 5.4: Object adapter as defined by [GoF95]

The `Client` on the left side wants to work with the component on the right side (the `Adaptee`). In order to promote clean software development, the client is written against an interface. This interface is called `Target` here and offers the services that the client expects. The `ObjectAdapter` implements this interface. It also maintains a reference to an instance of the `Adaptee`. Upon a service request, the adapter forwards all calls to the instance of the `Adaptee`. This means the adapter essentially “wraps” the `Adaptee` and delegates incoming method invocations to it. In contrast to the object adapter pattern, the class adapter pattern follows a slightly different approach. Instead of having a reference to an `Adaptee` instance, the class adapter uses inheritance to achieve its goal. The original design of this pattern used multiple inheritance which is not directly supported in Java. However, the basic idea remains the same, the class adapter also forwards incoming calls to the corresponding `Adaptee` method. The difference is just that this time the adapter does not maintain a reference to the `Adaptee`, because it has inherited all the methods from it. A Java implementation of this pattern would have to simulate the multiple inheritance, which is possible in a rather simple way: the target interface has to be defined as a Java interface and not as an abstract class. The Java-version of the pattern is shown in the following UML class diagram.

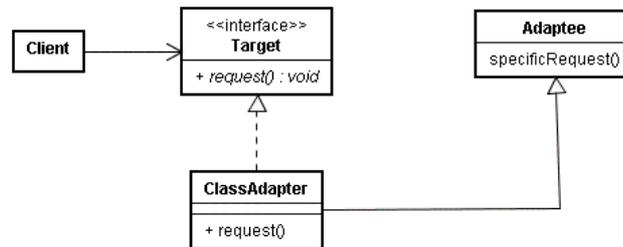


Figure 5.5: Class diagram of the class adapter.

As clarified by the figures, both patterns can easily be implemented in Java. However, they both have their advantages and disadvantages, especially when it comes to more complex target interfaces, as we shall briefly discuss in the following subsection.

5.4.2 LIMITATIONS OF THE GOF ADAPTERS

At a first glance, the adapter pattern seems to be the ideal candidate to use in a reuse-oriented development and it is recommended for this purpose by the GoF. But, the two versions of the pattern that we described above, bring along some unwanted disadvantages which hinder their usage for some more complex interfaces. First and foremost, the class adapter requires the adapter itself to inherit from the `Adaptee` class which makes it impossible in Java to inherit from any other class. This is a widely known issue which in general makes the object adapter the preferable solution. But there exists another issue that considerably limits the use of the two adapter. All the code that only involves primitive data types, `Strings`, or `Objects` can indeed be reused without any problems. But, all the classes that reference objects of their own type cannot be adapted with the above mentioned pattern variants. The following extract from the interface of a `BinaryTree` will illustrate this problem.

```

public interface BinaryTree {
    public BinaryTree(int value, BinaryTree left, BinaryTree right);
    public BinaryTree getLeft();
    public void setLeft(BinaryTree bt);
    public BinaryTree getRight();
    public void setRight(BinaryTree bt);
}

```

The critical elements of the code fragment above are the methods `setLeft` and `setRight` and their corresponding getter methods `getLeft` and `getRight`. The same problem also holds true for constructors if they contain self-referencing parameters. The following sketch illustrates the problem with the setter-methods more closely:

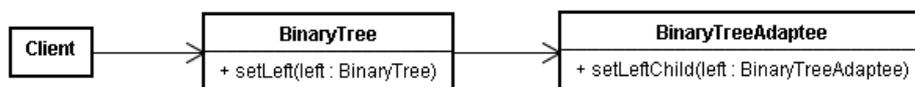


Figure 5.6: A situation in which an object adapter would fail.

As visible in the sketch, the `BinaryTree`'s (i.e. the adapter's) set-method expects a parameter of type `BinaryTree` which would be delivered by the client and normally be passed on directly to the `BinaryTreeAdaptee`. Of course, the latter object (and its method) only know its own type and has no knowledge of the existence of the adapter class and of the fact that a `BinaryTree` instance would be delivered in this case. Thus, an incorrect parameter type would be passed to `BinaryTreeAdaptee` and the adaptation would fail in this situation.

This issue is rather easy to solve, as we will show for the general case in figure 5.7. If the method `anotherRequest` in our so-called `ManagedAdapter` is called with an instance of itself, it simply forwards the call to the `anotherSpecificRequest` method and has to make sure that the `ManagedAdapter` is replaced with that `Adaptee` instance that was created for this `ManagedAdapter`. This explains why each `ManagedAdapter` should have a `getAdaptee` method as demonstrated in the figure.

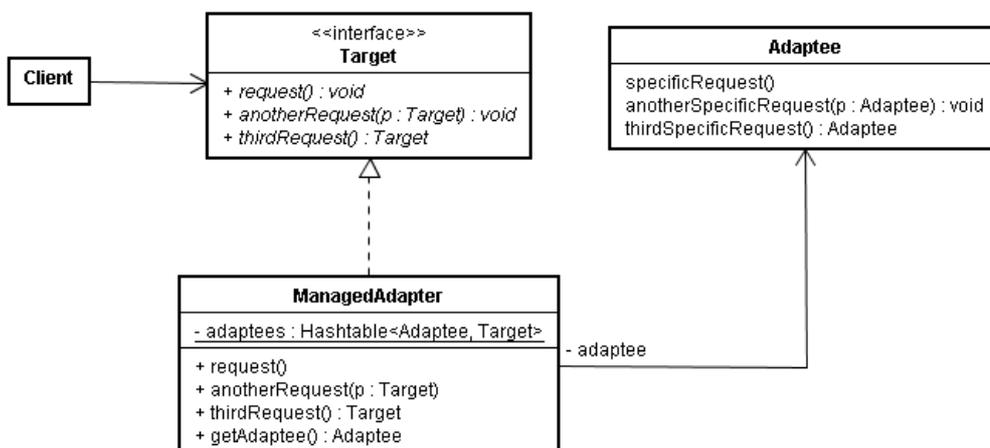


Abbildung 5.7: `ManagedAdapter` that overcomes the problems of the GoF adapters.

More challenging, however, is a solution for the case when a return value has the type of the adapted class (i.e. for getter-methods). We have developed the following solution for this issue. As shown in figure 5.7, the `ManagedAdapter` stores all adaptees it creates in a `Hashtable` where `Adaptee` objects are used as the key. Once an `Adaptee` object is returned by a method of the class `Adaptee` the appropriate instance of the `ManagedAdapter` can be looked up in the `Hashtable` or a new one can be created if there is none. However, one limitation is obvious with this solution if it is applied to Java programs. Since one reference to a `ManagedAdapter` object is always stored in the `Hashtable` the garbage collector would never be able to recognize such an object as unused and delete it. And since no explicit possibility to delete objects is available in Java, an object could not be deleted. For this reason we consider this solution only useful in the context of Extreme Harvesting where an object is only used for testing and where programs typically run for just a few moments in an isolated virtual machine. For practical use it seems to be better to refactor the `Adaptee` in order to fully adapt it to the interface of the target. Another solution for this problem is inspired by C# where an `IDisposable` interface offering a `dispose` method is used to delete objects. If the `ManagedAdapter` implements such a method, clients that are aware of this feature would be able to delete adapter objects by calling their `dispose` method.

5.4.3 PARAMETER PERMUTATOR

After explaining how potential candidates can be found and adapted for the testing process, we have to explain how situations like the one in figure 5.3 can be resolved automatically, i.e. how the most likely counterpart for a desired method can be found in a reuse candidate. In other words, the new challenge to be addressed at this point is finding the “correct” way of mapping the operations of the desired component to those of candidate components. Since the names of operations and parameters are at most an indication of their meaning, but in practice have no impact on the semantics of the operations, all possible type-correct permutations need to be considered. The basic idea is to execute the test case associated with the query until a mapping for the adapter which passes all tests is discovered. One approach to do this would be to create adapters for each potential mapping, but since for each permutation a new adapter has to be created, compiled, and executed, a large overhead would be involved. Thus, we have implemented a more efficient solution that uses Java’s reflection capabilities. The idea is to compile the adapter once, but to interpose a so-called `Permutator` object that uses a new mapping in each test run until all test cases are passed without error. The basic flow of this process is shown in the following sequence diagram.

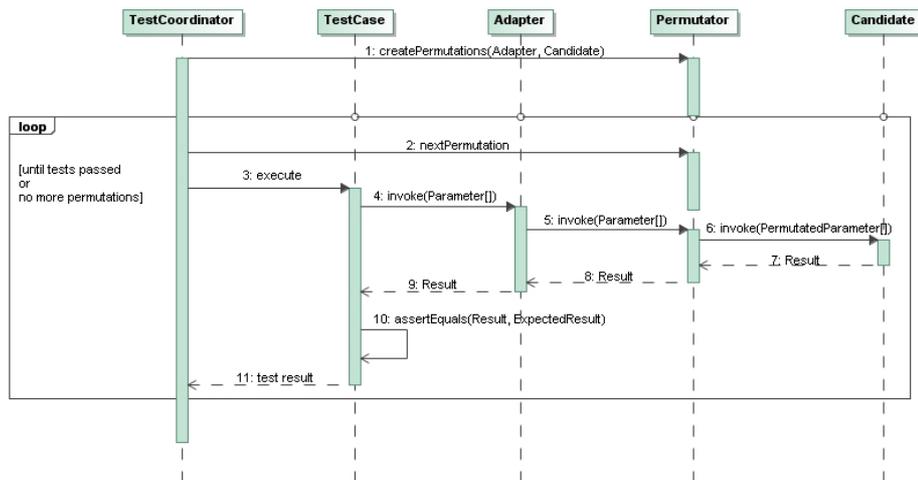


Figure 5.8: Sequence diagram of the testing process.

The `TestCoordinator` object on the left is responsible for managing the whole testing process. It initiates the `Permutator` object and lets it create all possible permutations for the method and parameter mappings derived from the interfaces of the `Adapter` and `Candidate` objects. After that, it starts the permutation and testing cycle by executing the `TestCase`, which is a normal JUnit test case. Since the adapter is adjusted to the calls of the test case, no changes need to be made to the code of the test case. The `Adapter` object, in turn, forwards the parameters and the ID of the invoked method to the `Permutator` object which looks up the appropriate permutation for the current cycle and invokes the candidate with the appropriate internal mapping. This loop is executed until either the test case is completely passed or no further permutations are available. The former case obviously occurs for semantically acceptable reuse candidates while the latter result indicates that the candidate is for some reason not reusable in the given context.

The creation of the permutations in the `Permutator` is a two stage process. First, it is necessary to determine whether two method signatures are identical and thus a potential mapping from a required method to a candidate's method can be established. Once this is finished, all feasible parameter orders have to be found for each mapping. For a better understanding of this concept consider the following example where the method calls from a `RequiredCalculator` need to be mapped to the relevant calls of the `CandidateCalculator`:

```
class RequiredCalculator {
    int add(int i, int j) {}
    int sub(int i, int j) {}
}

class CandidateCalculator {
    int something(int x, int y) {}
    int sub (int a, int b) {}
    int add(int x, int y) {}
    int no(int a, int b, int c) {}
}
```

First, all feasible mappings from the methods in the `RequiredCalculator` to the `CandidateCalculator` are established as follows:

```
add -> something    add -> sub    add -> add
sub -> something    sub -> sub    sub -> add
```

Once all method mappings are established, we can start creating all feasible parameter permutations for each method mapping, i.e.:

```
add(i, j) -> something(i, j)          add(i, j) -> sub(i, j)
add(i, j) -> add(i, j)
add(i, j) -> something(j, i)          add(i, j) -> sub(j, i)
add(i, j) -> add(j, i)
...
```

The next step is to combine the mappings for `add` and `sub` without accepting mappings where a method from the candidate is used twice, i.e. a mapping `add -> add + sub -> add` is to be avoided. Thus, the following combinations remain acceptable:

```
add -> something + sub -> sub    add -> something + sub -> add
add -> sub + sub -> something    add -> sub + sub -> add
add -> add + sub -> something    add -> add + sub -> sub
```

Of course, these mappings have to be combined with the parameter permutations and in our example this yields four permutations per mapping, e.g.:

```
add(i, j) -> something(i, j) + sub(i, j) -> sub(i, j)
add(i, j) -> something(j, i) + sub(i, j) -> sub(i, j)
add(i, j) -> something(i, j) + sub(i, j) -> sub(j, i)
add(i, j) -> something(i, j) + sub(i, j) -> sub(j, i)
...
```

Once all potential mappings have been established, the `Permutator` is able to “wire” the incoming parameters from the `Adapter` to the appropriate parameter in the `Candidate` as specified in the current permutation.

5.5 DEPENDENCY RESOLUTION

The process introduced above only functions for components that carry their whole functionality in one class. However, software systems today are typically so large that they have to be divided into a number of classes and modules that have dependencies on one another. However, Java and most other common programming languages today do not make required interfaces explicit in the source code, at least not very precisely. Consider full package imports such as `java.util.*`. It is neither clear which classes from that package will be needed later during the execution of the class nor is it apparent where to obtain the package. Java virtual machines typically search for importable classes in its classpath during compilation and execution and would successfully find everything in the standard util package. However, as soon as external packages come in to play, there exists no standard that would guarantee that the package could be downloaded from a specific URL, for example. Hence, Java developers often have to deal with missing dependencies as already discussed in the context of the search use cases in section 5.1.3. A powerful dependency resolution approach is thus a critical element of an effective Extreme Harvesting implementation. This is evidenced by the fact that in our Merobase index only about 44% of all Java source classes from the open web and only about 15% of all source classes from version control repositories could be compiled with the standard Java (J2SE) classpath. Or in other words, about three quarters of our Java source files have dependencies that need to be resolved before a class can even be compiled.

During the development of our harvesting solution we found the following heuristics that are helpful to mitigate this problem. However, although these ideas are all rather straightforward they also all contain some limitations which means that they will not always be successful and will under given circumstances even collide with each other. In other words, these heuristics can create an inconsistent classpath which causes the compilation to fail. If this situation occurs, there is in the end no other solution than a simple trial and error approach that tries to compile and run each possible solution similar to the permutation approach from before. However, such an expensive approach might not always be worthwhile in practice. Nevertheless, the following table contains the heuristics we propose as well as a discussion of their likely benefits and limitations.

Heuristics	Advantages	Limitations
Extend classpath with common JAR files	Easily applicable for standard libraries of other Java editions such as J2ME or J2EE and some well-known libraries such as Apache commons etc.	Different required versions of the files might impose a risk for this approach. Furthermore, some of these libraries might conflict with one another. Prominent examples for Java are loggers and XML parsers. Potentially large libraries have to be delivered with the classes.
Extend classpath with libraries from project	Helpful for files from CVS/SVN if libraries are indexed as well.	Not applicable for HTTP-based files. Libraries must be delivered with the class.

Heuristics	Advantages	Limitations
Explicitly search for libraries containing the missing classes in an engine such as Merobase	Promising for every class potentially published somewhere in a library.	Creates additional load and traffic on the search server. Candidates are not always decisively identifiable.
Derive URLs of missing classes from their names	Simple and easy to implement technique especially promising for classes that belong to the same package as the one that was compiled.	Often classes from different packages or even projects are required that might not be findable on the current server.
Search missing classes in Merobase	Potentially very powerful technique that should be able to find most dependencies.	The right candidate is hard to identify which is why this technique can become very complex and expensive, especially if adaptation heuristics are taken into account.
Create empty stubs for missing classes	Useful for Exceptions and Interfaces that could not be found with one of the other methods.	The originally intended functionality is lost.

Table 5.5: Potential heuristics for resolving missing dependencies.

As the table illustrates, we have developed a number of heuristics that come close to the dependency resolution capabilities of a human. However, they ultimately struggle with the same problems as human developers since it is often not clear whether the correct version of a required dependency has been found. This issue, though, can only be solved by trial and error, but it is a major question whether the required effort in processing and implementation time is worthwhile.

5.6 CLASS ENSEMBLES

As we explained in section 2.4, the optimal granularity for reusable *component* is an issue that has probably been discussed since components were invented. And it looks as if there is no optimal answer to this question since most mainstream component-based development approaches use a hierarchical component model that allows recursively composed components. This is an important issue for a practical implementation of Extreme Harvesting or any other search technique in software component markets. While most of the examples that we have shown so far can be built with just one class, it is obvious that this does not scale up to more complex functionality. The `Blackjack` component, shown in the following figure, can be considered as a simple example of such a case. With the help of our dependency resolution algorithms introduced above we were able to discover it with one simple test case.

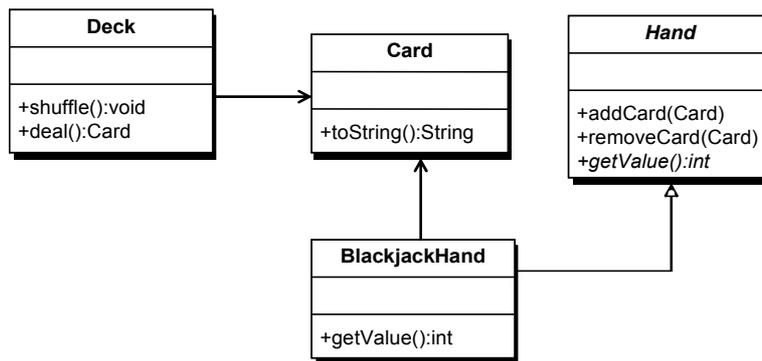


Figure 5.9: Simplified class diagram of a harvested Blackjack component.

However, the literature does not contain any guidance on how to derive a query for such a component or on how the results should be returned from search engines that are specialized on primitive components consisting of single files. To exchange queries and results, it makes sense to use the standardized representation of a class diagram as the query (i.e. XMI data) and return the results packaged according to another standard such as the RAS [OMG04b]. Depending on the implementation language and the used IDE it could also make sense to package the results as JAR files, or maybe as Eclipse or even IDE-independent Maven (maven.apache.org) projects.

In addition to these technical issues, the question of how best to search for such class ensembles still remains. Currently, we see two possible solution strategies. The first one would be to sequentially search for each class described in the class diagram and to build the ensemble by collecting all the smaller ones. This could either be done in a manual approach that implements a design such as that the above in the usual way e.g. a test-driven development process. In other words, as e.g. recommended by [Lar05] the least connected class of the design (in the case of the example this would be Card) is selected first, a test case is created for it and the harvesting tool is used to search for an appropriate implementation. Then the second least connected class (Hand in this example) is chosen and so on until the whole system is implemented. We will present a little case study that demonstrates its power in section 7.5.

The other approach that looks promising in order to save manual labour is to speculate on the capability of the dependency resolver. It might be able to discover required classes if test cases for the most connected class (i.e. the one with the most dependencies) from the design are created and executed first. Once a candidate for these test cases is found it is possible that the dependency resolver might find the other required classes without additional manual effort. This idea was applied for the above blackjack example and yielded two implementations from the web without further human intervention. This comes already quite close to the approach that is envisaged in KobrA [Atk02] and would use the specification of the upper-level component to find the complete ensemble in one go. Although we were able to find a picture viewer, an arkanoid game and other applications based on speculative searches by means of such a dependency resolver, one inherent problem could not be solved within this dissertation – the problem that the component concept that is incorporated in KobrA is not implemented in today's programming languages. We discuss this challenge in more detail in section 6.2.

5.7 IMPLEMENTATION

As we already pointed out at various points during this dissertation, an appropriate tool is a very important requirement for the application of our ideas and the only vehicle to demonstrate that they are applicable in practice. Although the overall Extreme Harvesting process is straightforward and should be rather simple to implement, “the devil is in the details”, as is often the case. From the first standalone prototype, through various Eclipse-based versions to the current sophisticated (and safe) client-server implementation of Merobase, the implementation of each version had its special challenges. Whether it be the execution of the Java compiler and the analyses of its error messages, the correct configuration of the Java security manager, the use of Ant tasks to simplify compilation and testing, or the difficulties of Eclipse programming (see [Jan07] for some more insights on that), many tasks often caused unexpected difficulties due to weak documentation. However, we do not want to go into all implementation details at this point and thus limit ourselves to the essentials in this subsection. The following figure summarizes the current structure of our system, which is organized as a classical 4-tier-architecture (as described e.g. in [Som06]):

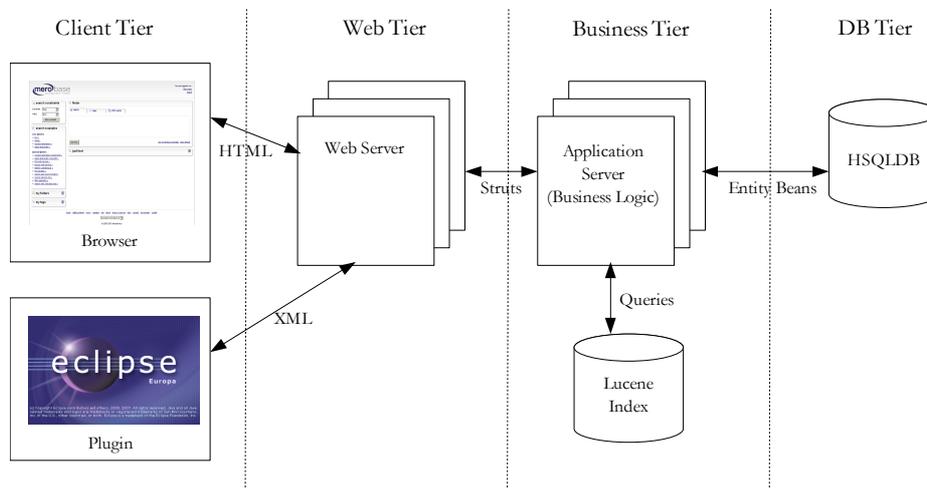


Figure 5.10: System architecture for Extreme Harvesting with Merobase.

The remainder of this chapter, covers the most important issues in the structure and implementation of our current client-server solution depicted above. Since it is of more interest for users of our approach, we want to discuss the ideas behind the Eclipse plugin first before turning to the backend implementation.

5.7.1 ECLIPSE PLUGIN

Related work has given strong indications that reuse seems to work best if queries are created automatically (or proactively) out of the developer's current working environment and reuse candidates are recommended in an unsolicited way [YeF05] & [McC07]. Thus, we also built a proactive reuse recommendation tool integrated in a common development IDE. Obviously, a successful implementation of this idea would immediately avoid most of the reuse failure modes discussed in section 2.5.2. The precondition for a successful reuse recommendation tool following this maxim, however, is that it is able to automatically generate queries from what the developer is currently doing. In

other words, such a tool should be able to trace the programming activities of a developer and automatically send off queries at appropriate points in time. Exactly this issue was a fundamental weakness of CodeBroker since it required its users to “actively” comment the source code they were working on. In other words, developers had to describe the code they intended to write in natural language before they starting to implement it. This was necessary to generate the queries for the LSI-based [Dee90] search system driving CodeBroker. However, in our opinion, much of the advantages of a proactive system are lost if developers have to “describe” their intentions to the system first. Since Extreme Harvesting and Merobase were developed with this issue in mind, we were able to optimize our plugin to avoid such problems. Our so-called component finder Eclipse plugin simply monitors the coding work of the developer and when some specified event occurs (to be described later) the interface of the code is extracted, and sent to Merobase as a search query. Since Merobase is able to parse (Java) code this can happen totally transparently to the developer. Reusable components can then be presented to the user within just a few seconds. The following figure illustrates how this functionality can be used in practice to carry out interface-driven searches on Merobase:

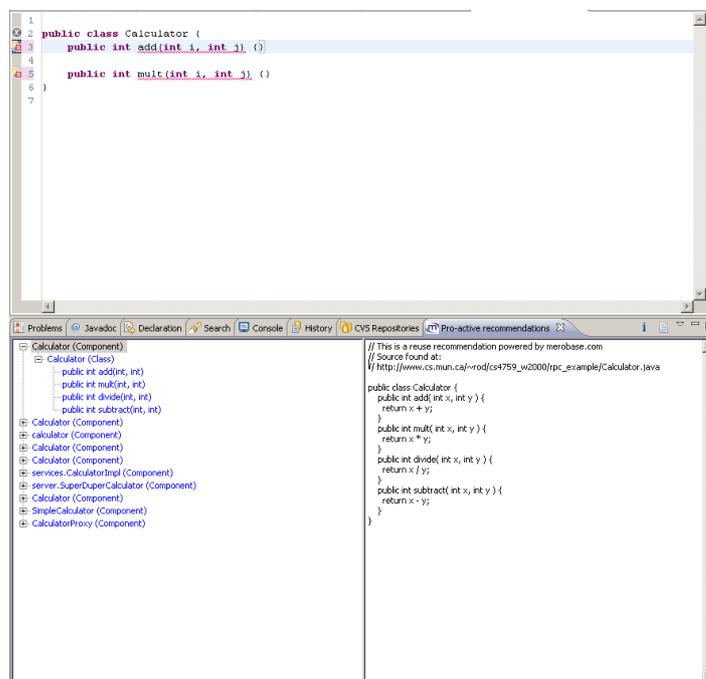


Figure 5.11: Our Eclipse plugin suggesting reusable candidates based on an interface-driven search.

As soon as a user adds a new method signature to the class he is editing in the area at the top of the figure, the plugin recognizes the new information, sends it to Merobase and provides the user with reuse recommendations that are likely to be useful in his context. It does not even matter that Eclipse complains about the missing return statements inside the method bodies. Once the lower segment of the window is populated, the user can browse the discovered components and study their code in the right hand part of the window. If a satisfying candidate is found, a double click on the candidate or one of its methods in the tree on the left will copy exactly the desired element to the correct place in the editor at the top.

Within the context of an agile development project that utilizes test-driven development (TDD), it is possible to go even further. In the purest form of TDD, developers do not start their development or design work with the class stub, instead they write a suitable test case before any production code is written. For the `Calculator` example from the figure above such a test case might have the form shown in the following figure. Thus, the key point in this variant of the process is that the class under test (e.g. the `Calculator`) does not even exist, which again leads to problem reports in Eclipse:

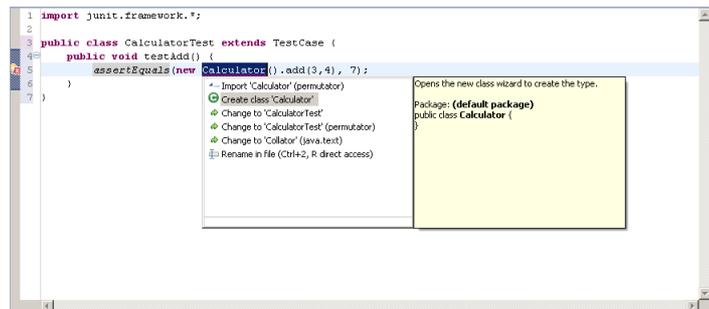


Figure 5.12: Using Eclipse's "quick fix" function to derive a class stub from a test case.

A test-driven development purist would now use the “quick fix” function of Eclipse and let it generate the missing class and its methods one after the other, which might at least save some time in comparison to the manual creation of the stub. But our plugin is able to reduce the workload even further. As soon as the user has created a viable test case in the top window, the tool can start searching for matching components. The list of syntactically matching components is again displayed in the bottom left-hand window proactively after just a few seconds. However, using a test case as the search query offers another significant advantage as it allows to fully implement semantic retrieval based on Extreme Harvesting and thus to directly and fully test all reuse candidates in the result list using the defined test case:

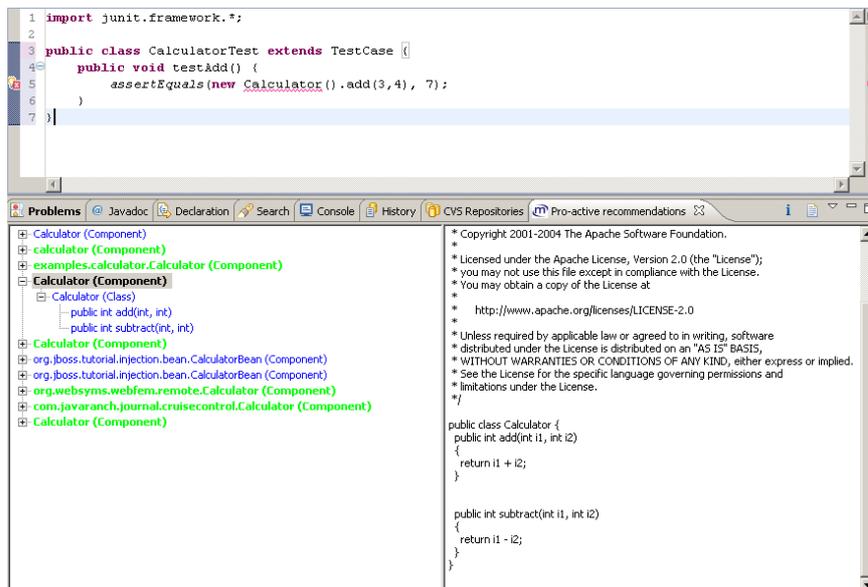


Figure 5.13: Test-driven reuse proposals by the Eclipse plugin supporting Extreme Harvesting.

As soon as a component is successfully tested, its name is written in bold face and green as shown in figure 5.13. To provide this ability to deliver results as soon as they become available, the plugin regularly polls the server every 15 seconds in order to check if new results are available (since the underlying Opensearch protocol is based on HTTP). Often, studying the code of discovered components gives the user more insight into the required behaviour of the component, and if none of the initially proposed candidates is exactly suitable, he can extend the JUnit test definitions and continue the “harvesting” process in an iterative way until either an acceptable candidate is discovered or no more candidates are available. A simple case study using this plugin is described in section 7.5. For details on the client-side implementation of the plugin we refer to [Jan07]. An overview of the architecture on the server-side is given in the next subsection.

5.7.2 SERVER-SIDE IMPLEMENTATION

This subsection gives a brief insight into the system structure on the server-side. Basically, as already indicated above (in figure 5.10), Merobase is implemented as a classical 4-tier-architecture comprising the client tier with the web browser or the Eclipse plugin just explained and the three server-side tiers. These comprise the web tier, the application server (JBoss), containing the business logic and the Lucene index and finally the database that stores the EJBs. As far as the search engine functionality of Merobase with the web front end is concerned, current dual-core servers (as of October 2007 with 2*1 GHz, 2 GB RAM, 200 GB HDD) are easily capable of satisfying a user request in typically less than five seconds. Merobase began as a web application that had to be used through an HTML-based interface in a web browser in early 2006. For programmatic access to our search engine we later added an API and slightly extended the Opensearch format [Cli07] developed by Amazon for that purpose. It delivers an XML feed that can be parsed and processed by tools such as our Eclipse plugin.

The architecture described above is not sufficient for Extreme Harvesting, however. When large numbers of files have to be compiled and tested, significant processing power is needed and security risks caused by the execution of unknown code are introduced. Thus, it makes sense to externalize this task to one or more supporting machines. This, might be sufficient in terms of performance, but not in terms of security and thus we developed a solution based on virtual private servers (VPS) as shown in the following figure:

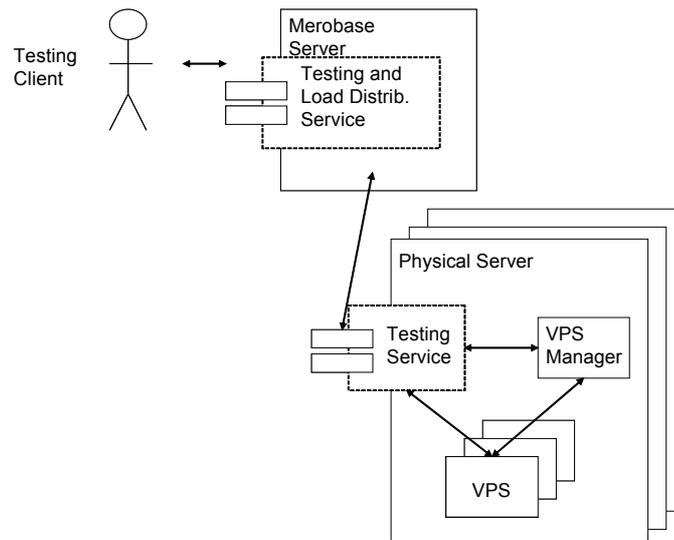


Figure 5.14: Harvesting system architecture.

There are various open source (such as Xen developed by the university of Cambridge) or commercial (e.g. Virtuozzo by SWSOft) solutions available that allow an arbitrary number of isolated virtual machines to be set up on a physical server. Today, these are already in widespread use by hosting companies that offer virtual servers to their customers. We use this approach to provide the security needed to execute test cases and unknown code from the Internet and to provide sufficient control over the resources of the server. Although we haven't experienced any problems with malicious code in our experiments so far, it is likely that such attempts will occur as soon as such a service becomes publicly available. Each running VPS instance is periodically checked by the VPS Manager, and as soon as it becomes aware of any problems (such as a system not responding anymore), it shut down the running instance automatically and powers up a new one. Such a switch usually takes about 60 seconds until the new VPS is fully operational.

We run a Tomcat web server on the VPS which provides a web service based interface to the testing service to the main Merobase server. The latter, sends test cases and a list of files to test to the VPS which then performs the compilation, dependency resolution, permutation and test case execution steps as explained in the previous parts of this chapter by utilizing further open source tools such as JUnit and Ant. The test results are collected on the main server and are sent back to the Eclipse plugin as soon as this polls the server or to a user of the website by email once the specified number of results have been tested.

6 PROCESS INTEGRATION

*Organ transplants are best left to professionals.
Organ transplants are best left to professionals.
Organ transplants are best left to professionals.*

...

-- Bart Simpson's Chalkboard

It is interesting that component-based reuse has been on everybody's lips for almost four decades, but there is still very little theory on how to apply it in common development processes, as discussed in section 2.6. In this chapter we thus describe how the ideas of this thesis can be used to integrate and foster reuse in today's software development processes. Given the sheer number of software development processes and methodologies we choose two representative examples – namely Extreme Programming [Bec99] and KobrA [Atk02] - to demonstrate how the findings and developments of this dissertation can be used to enhance everyday development practice. It should be straightforward from the presented processes to integrate these results into other methodologies – such as an Agile RUP as proposed by [Lar05] – as well.

Since the current generation of publicly accessible component repositories are almost all source code centric and offer only basic text-search capabilities, at the present time it is difficult to use them for more than just “code scavenging” [Kru92]. This practice involves copying and pasting of small code snippets into the system under development and is discouraged in many publications such as for example the Anti-Pattern Book [Bro98]. The argument against the reuse of such snippets is that it requires a lot of effort to find appropriate snippets and their use is more likely to degenerate the design of the system under development than to improve its quality. However, these snippets can certainly be useful if they are used as an inspiration for how to solve a problem, rather than as a way of avoiding the programming of the solution from scratch. With the techniques developed in this dissertation it has become possible to increase the granularity of reusable elements up to small components which can be selected based on their specification in the system design. However, although software reuse has been the subject of research for almost four decades, there is still no clear picture of when and how reusable components should be used in a development process. Even modern development methodologies contain few if any guidelines on how to select components based on their specification. In general, since reuse candidates will usually not match perfectly, a feedback loop is often necessary where either the design or the

candidate have to be adapted, as we have already explained in section . To our knowledge, this idea is currently best described in [Crn06].

Our own experience with reuse repositories indicates that the best kind of component search to use in a development process depends heavily on the point of time at which the search is performed rather than on the nature of the process itself. The earlier the point in a system's development process at which a search for reusable components is performed the less design work is likely to have been carried out. Hence, a general "speculative" search is more useful in early development phases and can feed back valuable information about available components and their interfaces into the design process. On the other hand, if a component search is carried out at a relatively late point in the development process, an interface- or even a specification-based search (the latter includes a semantic description as well) approach is required. Furthermore, if binary components or web services are to be the subject of the search, there is no source code and thus the search has to use interface descriptions in any case. Considering these differing requirements, a component search engine must be very flexible and none of the first generation code search engines is able to support them all. The next subsection will discuss how the advanced features of Extreme Harvesting can be utilized within test-driven (i.e. typically agile) processes.

6.1 REUSE IN TEST-DRIVEN PROCESSES

Iterative and incremental development and software reuse are both strategies for building software systems more cost effectively. Iterative (and especially agile) methods do this by shunning activities which do not directly create executable code and by minimizing the risk of user dissatisfaction by means of tight development cycles in functionality is implemented. Software reuse does this by simply reducing the amount of new code that has to be written to create a new application. Since they both work towards the same goal, it is natural to assume that they can easily be used together in everyday development projects. However, this is not the case. To date, incremental development and systematic software reuse have rarely been attempted in the same project. Moreover, there is very little, if any, mention of software reuse in the agile development literature, and at the time of writing, there is only one published reuse concept whose stated aim is to reinforce agile development. This is the so called "agile reuse" approach of McCarey et al. [McC07].

The reason for this lack of integration is the perceived incompatibility of incremental approaches and software reuse. Whereas the former explicitly eschews the creation of software documentation, the latter is generally perceived as requiring it. And while agile methods usually regard class operations (i.e. methods) as defining the granularity of development increments, reuse methods typically regard classes as the smallest unit of reuse in object-oriented programming. As a third difference, reuse approaches tend to be more successful the "more" explicit architectural knowledge is reused (as in product line engineering), whereas agile development methods employ as little explicit architecture as possible. At first sight, therefore, there appear to be several fundamentally irreconcilable differences between the two approaches.

McCarey et al. suggest a way of promoting reuse in agile development through so-called “software recommendation” technology. Their “agile reuse” tool, RASCAL [McC07] is an Eclipse plugin which uses collaborative and content-based filtering techniques [Bae99] to proactively suggest method invocations to developers. Although the concept of RASCAL fits well into the agile spirit of providing maximum support for “productive” activities, there is nothing in the technology which specifically ties it to agile development. The approach embodied in RASCAL can just as easily be used with any other development methodology that produces code, including traditional heavyweight processes. Moreover, the approach has the same fundamental weakness as other repository-based approaches – the quality of the recommendations is only as good as the quality (i.e. the size and the precision) of the code repository that is used to search for components. The version of the tool described in [McC07] is clearly a prototype, but McCarey et al. do not present a strategy for solving this important problem. Moreover, although RASCAL showed impressive performance for the limited domain of Swing invocations, it is not clear whether this technique will work for other domains with repositories containing many more classes that have much lower usage frequencies.

We believe the core challenge of agile reuse lies in developing a reuse strategy that complements the principles of agile development and offers a way of promoting reuse in tandem with the key artefacts and practices of agile methods. In other words, we need to find a way to seamlessly integrate Extreme Harvesting into agile methodologies. Typically, tests are used as the basic measure of a unit’s semantic acceptability. Once a code unit passes the tests defining its required behaviour, it is regarded as “satisfactory” for the job in hand. Usually the code to satisfy the tests for a unit is implemented by hand. However, there is no specific requirement for this to be so as we have shown in this dissertation so far.

6.1.1 AN EXTREME PROGRAMMING EXAMPLE

As mentioned above, our approach most obviously fits with agile approaches for software development, since these normally also involve the definition of test cases prior to the attainment of implementations. The creation of test cases to evaluate software units is one of the fundamental tenets of Extreme Programming [Bec99] – in fact, they are usually defined before the units they are intended to check.

We assume that the reader is familiar with other fundamental principles of Extreme Programming such as the four values of communication, simplicity, feedback and courage and the many recommended practices. For further details we refer to [Bec99], for instance. The test-driven nature of XP requires in particular that unit tests be written for a software unit before the code itself. These tests are used as the primary measure for completion of the actual code. The maxim is that anything that can’t be measured simply doesn’t exist [Bec03] and the only practical way to measure the acceptability of code is to test it. To illustrate how test-driven development works in practice let us consider a small example. We choose the `Movie` class that, together with a `Customer` and a `Rental` class, forms the initial version of the well-known video store example in Martin Fowler’s refactoring book [Fow99]. This class is required to offer one constructor and three methods with the following signatures:

```
public class Movie {
    public Movie(String title, int priceCode)
    public String getTitle()
    public int getPriceCode()
    public void setPriceCode(int priceCode)
}
```

Please note that we only presented the full interface of this class here in order to facilitate a better understanding. Following the recommendations of Beck [Bec03], the XP development cycle would normally be applied iteratively, driven by the following to-do list:

- ◆ Create object with title and price code
- ◆ Retrieve title
- ◆ Retrieve price code
- ◆ Change price code

The basic idea is to define tests to check that the constructor works correctly in tandem with the retrieval method. This can be done by using one combined test or using a separate test for each retrieval method. In this example we choose the latter since it is the more realistic for larger components. First a JUnit [Bec99b] test case is created for the retrieval of the movie's title:

```
public void testTitleRetrieval() {
    Movie movie = new Movie("Star Wars", 0);
    assertTrue(movie.getTitle().equals("Star Wars"));
}
```

In practice, test cases would probably be more elaborate (for example, they might follow the principle of triangulation [Bec03]) but for sake of simplicity we have decided to stay with the most simple example. This is enough to convey the core idea. In the next step, a stubbed out version of the `Movie` class (similar to the signature above) with just the constructor and the `getTitle` method is generated (TDD purists typically use the quick fix function of Eclipse) and is made to compile. After this, the test case and the stub are compiled, and the test is run to verify that a red bar is obtained from JUnit. Once the failure of the test has been checked, the stub is filled with the simplest implementation that could possibly work, and the test is re-run until a green bar is received from JUnit. The to-do list is then updated accordingly:

- ◆—Store title and price code
- ◆—Retrieve title
- ◆ Retrieve price code
- ◆ Change price code

The same process is then applied to the next method on the to-do list. First, a test case is defined to check the functionality of the new method in relation to the already implemented code, a new method stub is added to the existing version of the class and the tests are executed to check that the new one actually fails:

```
public void testPriceRetrieval() {
    Movie movie = new Movie("Star Wars", 0);
    assertEquals(movie.getPriceCode(), 0);
}
```

The stub for the new method is again filled out with minimal implementation and the test is re-run until a green bar is received from JUnit. The to-do list is then updated again accordingly:

- ◆ ~~Store title and price code~~
- ◆ ~~Retrieve title~~
- ◆ ~~Retrieve price code~~
- ◆ Change price code

The last method of the class is then implemented in the same way and finally, the to-do list is completed.

6.1.2 EXTREME REUSE

As explained above, the basic idea behind our notion of agile or extreme reuse is to use test cases that are developed as part of the normal activity of Extreme Programming as the basis to search for suitable existing implementations. However, to achieve the maximum benefit of our specification-based reuse approach, it would be necessary to define the test definitions for all methods and thus the unit's complete interface in advance. Then, if the harvesting is successful, the only additional step would be the invocation of our reuse tool. All additional implementation work usually involved in building a component from scratch would thus be avoided. However, this approach suffers from some drawbacks. First, it requires an unnatural processes from the point of view of agile developers who are used to iteratively developing a unit under test as explained in the last subsection. Thus, the full specification-based approach is probably more suitable for an agile version of the RUP, as recommended by [Lar05], since the RUP recommends that a system design be created before the test cases are prepared on the basis of UML class diagrams. We discuss this idea in more detail in the next subsection when we will present an agile version of Kobra. The second issue arising in this context is the difficulty of anticipating the content of the reuse repository, which has plagued all reuse approaches right from the beginning. As we mentioned before, the more complex a component becomes, the lower the probability of finding a reusable component that exactly matches its specification [Sam97]. Thus, all reuse approaches developed so far include a negotiation phase where the specification is changed to conform to the retrieved candidates and/or a glue coding phase where the components are adapted to fit into the system design (cf. e.g. figures 2.12 and 2.14 in section 2.6).

Given our Extreme Harvesting approach and our proactive Eclipse plugin, we are able to reduce this overhead significantly if we integrate it in a straightforward manner into a test-driven process, as sketched in the last subsection. The general idea is to provide immediate feedback on each new test added to the test case class. Thus, we optimized our plugin accordingly in order to support the following process demonstrated by means of the movie example from above. The first action where our plugin can intervene occurs when the developer is about to create the new test case, as shown in the figure below:

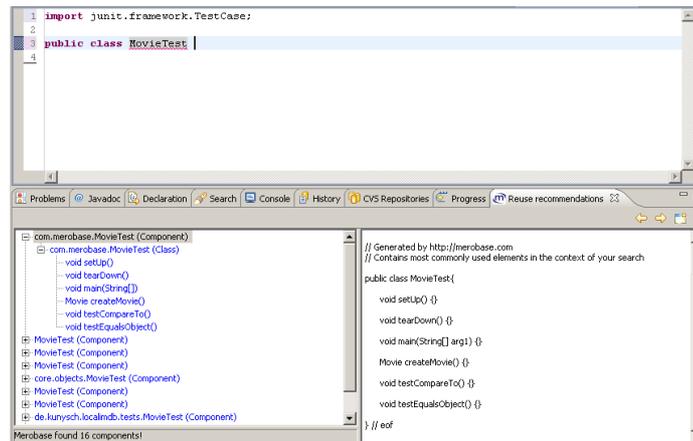


Figure 6.1: The Eclipse plugin recommends potentially useful methods for MovieTest class.

The proactive search mechanism queries the Merobase index to find out whether other developers have created a `MovieTest` class before. In this case there are actually 16 results available. With a little luck, we might find a test case that corresponds well with our to-do list and could avoid the work of deriving our own test for it. Out of these 16 results, our plugin is able to derive the most often used methods and presents them as the first result. However, the test methods written by other developers are not particularly useful at this point. And even a brief overview of the results presented does not reveal any test case that would test the functionality we require. Thus, we continue to add the first test method to our test case as designated in our to-do list which then makes it possible for our plugin to generate a new list of recommendations. This time, it realizes that we specified functionality of another class within the test case and thus presents a list of `Movie` classes that are potentially capable (based on a syntactic analysis) of delivering the needed functionality as shown below:

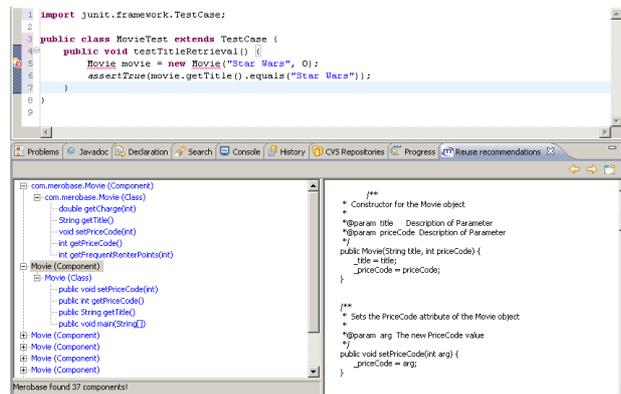


Figure 6.2: The plugin presents reusable candidates that are likely to offer the required functionality.

Just another click to trigger the testing of the presented candidates delivers the following list where successfully tested candidates are printed in green and bold face:

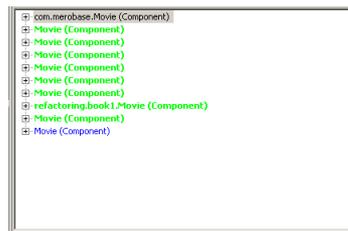


Figure 6.3: List of results that actually deliver the required functionality.

This step obviously is the key of the whole test-driven reuse process. If no syntactically suitable candidates are found, we could change the interface of the class under test. However, the probability of finding a usable candidate would obviously be rather low and it might make more sense to focus on programming from scratch. Since we have actually found a large number of matching candidates, we could now take advantage of the additional information that these candidates bring along. We could either create a `Movie` class and copy the required constructor and method from one of the candidates and continue with a test case for the next requirement on our to-do list or we could choose the candidate that seems to be closest to our to-do list and integrate it completely into our project. After all, the latter alternative bears the risk of significant adaptation effort since it is not guaranteed that the interface of this candidate fits into the system under development and the test cases that are to be defined for it.

Thus, we believe, it makes sense to create a hybrid version of both approaches and add one test method at a time and let Extreme Harvesting present all candidates that successfully pass the test at each point. As long as there are more candidates available, another test method can be added with the interface that is desired for the system and the ones that are offered by the candidates. In this way, it is feasible to incrementally add more and more functionality to be checked to the test case until there is either a number of candidates that offers the full functionality or there are no more appropriate candidates. In the former case, one of the candidates can be selected to be integrated into the system and in the latter case, one of those candidates that was left over before the last test method was added should be chosen. It can be integrated into the system and the additional functionality added manually.

6.2 COMPONENT-DRIVEN DESIGN WITH KOBRA

It follows from the previous discussions that the ideal methodology for use with our Extreme Harvesting approach is one which integrates model-driven and component-based development approaches and allows them to be applied within the context of an agile development process. Several methods claim to do this, such as Catalysis [Sou98], UML Components [Che00] or even an agile variant of the RUP, as described e.g. by [Lar05], although the latter has certainly weaknesses in being component-based. We believe the method which currently offers the cleanest integration of these two paradigms is Kobra [Atk02]. The essential difference between Kobra and other mainstream methods that claim to integrate components and models is that in Kobra all UML diagrams (not just their contents) are organized around the logical components in a system. Thus, components are considered in analysis and design (at the level of the platform independent model or PIM) rather than just at the implementation (i.e. the

platform specific model or PSM) level as is often the case in other methods. KobrA uses the three different perspectives we have depicted in figure 2.6 to specify the structure, functionality and behaviour of a component.

As long as a KobrA component is implemented by just one class, it is relatively straightforward to translate a KobrA component specification into a query for Merobase and Extreme Harvesting. If appropriate UML tools (such as Omondo or Together that can be run inside Eclipse) are used, even a proactive search on KobrA's structural specification (the interface of the class specified in a UML class diagram) becomes feasible since the tools are able to generate class stubs usable as input for Merobase from their class diagrams. KobrA's behavioural specification, which is typically captured in state diagrams, can easily be turned into test cases (see e.g. [Kim99]). KobrA, in its latest version, aims to capture the functional specification of a component in OCL [War03]. Although at present it is not feasible to use OCL as the semantics description for our search approach, there is research underway that aims to executing OCL descriptions for testing (see e.g. [Bri01]) or derive test cases from them. In summary, a KobrA specification can be turned into an Extreme Harvesting query with relatively little effort and that should be automatable in the not so distant future. Moreover, this effort would usually have to be performed in any case in order to finally test the component once it is implemented.

However, as stated before, KobrA defines a hierarchical component model which is capable of condensing a number of smaller components into a larger one. Although this is, in principle, similar to packages in common programming languages, packages are not sufficient to implement this idea since KobrA components can not only package other components, they are also facades [GoF95] at the same time. Unfortunately, the latter is not possible in today's programming languages, which makes it very difficult, if not impossible, to apply the KobrA component model to them in a simple way since important KobrA concepts are not directly supported. Although we are not aware of any work in this direction, we believe it should be feasible to mimic the behaviour of nested KobrA components in Java by means of inner classes. But this idea is not thought through yet and also comes with two important drawbacks. First of all, the composition of these Java classes to new components would involve copying subcomponents into the source code of their parents which is fundamentally against the rule that a component must be independently deployable and furthermore it would make this approach highly confusing after just one or two composition levels. The second problem with this approach at the present time is that it is not widely known, let alone widely used and thus it is not possible to find any existing components that adhere to this principle. Unfortunately, it is also not feasible to adapt existing packages of class ensembles into KobrA components as described in section 5.6. This would require extensive reverse engineering to extract the structure of the facade inherently hidden in the interfaces of the classes in the ensemble, and thus could only be carried out by a human.

Beyond these technical issues, the design-first approach just described is not the way in which product architectures are developed in other engineering disciplines. When developing a new product in well established domains such as automobiles or computer hardware, engineers start off with a good idea of what kind of components are available and what kinds of architectures have been used in the past. The process of developing a new architecture for a new product is thus a highly iterative one, with ideas for possible architectures being developed hand in hand with the identification of possible components to realize them. Like the agile reuse approach described above, the basic process behind component-driven

design in Kobra should be highly iterative, and therefore requires a tight feedback loop between the component identification and component selection activities:

1. Model pre-existing components
2. Flesh out potential first cut design, reusing already identified components wherever possible
3. Model the interfaces of any new components required in the design
4. Search for new components matching these interface
5. If perfect matches are found for all new components, use them
6. If not, repeat from (2)

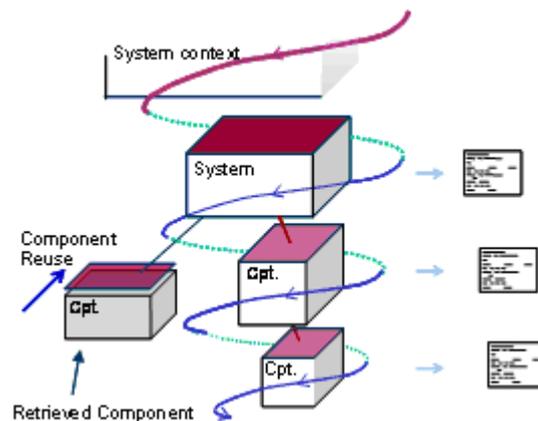


Figure 6.4: Iterative component modelling process in Kobra.

The key ideas behind this process are illustrated in figure 6.4. The central part of the figure shows a hierarchy of components. Each component is represented as a box whose top surface is the component specification and whose bottom surface is the component realization. According to the Kobra consistency rules each subcomponent's specification must conform to the realization of the component containing it. In other words it must conform to the containing component's architecture. The canonical development process is top down as indicated by the spiral arrows in the centre. However, the left hand side of the diagram illustrates that whenever a desired component specification has been created, it is possible to find and integrate an existing component rather than developing a new one from scratch. And this is where Extreme Harvesting comes in. It is the job of the Extreme Harvesting engine integrated into the development tools to find and present candidate components based on the component specifications and the associated test cases. It is even conceivable to use an extended version of our tool to find components at runtime. The developer simply defines the syntactic and semantic interfaces of the required component and the tool can search the web or a web service registry for components that match.

But the real key advance in this process, and the reason why we refer to it as component-driven design is that the architecture is always developed or evolved with regard to the specifications that are known to exist and have already been defined. As in other engineering disciplines, therefore, architects have a pallet of existing components in front of them when developing the architecture. This contrasts with the situation today where architectures are first developed independently and then an attempt is made to

retrieve components matching those required by the architecture. In component-driven design, feedback about the availability of components is provided as soon as a potential architecture to use them is modelled.

Large software repositories such as the one that we created for Merobase open a lot of exciting opportunities. One of them is the idea to derive design recommendations from commonly used elements found in the repository, as we described in section 5.1.5. Although this idea is not limited to KobrA per se, we will explain how to apply it in the context of component-driven design within KobrA in the following subsection.

6.2.1 SUPPORTING SOFTWARE DESIGN WITH INTERFACE RECOMMENDATIONS

Assume that early in the design phase of a system the necessity of a stack data structure is recognized. At this point, a developer might add a corresponding class without any further information to the system design to refine it later. Then, design is typically driven by interactions with other objects to define the interface of the stack component. But consider a system that would be able to (actively) recommend commonly used operations of a stack to the developer. This certainly has the potential to ease the development process and to reduce problems arising due to missing operation interfaces and, of course, it makes a giant leap towards realizing the idea of grounding a design on approved solutions. Although the current version is limited to unique classes, it is easily conceivable to extract dependencies as well and to form more complex design recommendations from that. Furthermore, given common code and design metrics such as fan-in and fan-out [Hen81] it should even be feasible to recognize bad designs and to exclude them from the recommendation process. And last but not least, it seems appealing to investigate the potential of integrating approaches that try to recognize design patterns (such as [Kel99]) to improve the quality of the derived design suggestions.

6.3 GENERAL DESIGN GUIDELINES FOR SUCCESSFUL REUSE

Endres and Rombach [End03], identify common object-oriented design guidelines like high cohesion, low coupling and encapsulation of the implementation as the prime prerequisite for successful reuse. In general, it seems reasonable to claim that adherence to general design guides and rules should also increase the chances of creating a reusable component or of finding one. During the development of our approach we collected a lot of additional informal knowledge that seems to increase the chances of finding reusable components with Merobase and/or Extreme Harvesting. We share this experience in this subsection in the form of small reuse idioms that are intended to simplify daily work with a reuse system based on a large repository.

Follow naming conventions

It obviously makes sense to follow the (Java) naming guidelines such as using imperative verbs for method names, nouns for class and attribute names etc.

Use common data types

Modern computer systems with a lot of cheap memory obviously made programming more convenient and thus some primitive data types are still widely used. According to our experience, it makes sense to

use double instead of float (e.g. `add(double, double):double` vs. `add(float, float):float = 190` vs. 67 results in Merobase), int instead of long, short or byte (results for the add example from before: 920, 503, 20, 19) and String instead of char whenever this is feasible.

Use void whenever possible

Method calls that perform some operation on a potentially larger data structure (such as `sort(int[])`) should return void and change the parameter object whenever possible to preserve memory, e.g. `quicksort(int[]):void` could be found 140 times with Merobase while `quicksort(int[]):int[]` delivers just 13 results.

Use exceptions

Exceptions should be part of the interface, i.e. they should be thrown by a method and not be handled internally since different users may have different requirements for exception handling.

Combine atomic functionality

Searching for atomic functions is often more promising than searching for complex operations. For instance, components that sort an array of integers from the largest to the smallest are rather hard to find, but components that sort from the smallest to the largest and provide a method to reverse the array are typically easy to find. Sometimes it might also make sense to split up classes if their cohesion is low since this has also a negative impact on their reusability. According to our experience, reuse is currently much more promising if it is used as a bottom-up approach putting together very small functional units. Top-down reuse requires more sophisticated recommendation technologies, as we will briefly describe in the next section.

Search the least-coupled elements first

A guideline which is valid for the implementation and testing of components should be taken into account when searching for reusable components as well – namely, move from the least-coupled to the most-coupled classes as recommended by [Lar05], for instance. Many problems disappear if this guideline is followed, as for example demonstrated in the small case study described in section 7.5.

7 EVALUATION

*Computers are magnificent tools for the realization of our dreams,
but no machine can replace the human spark of spirit, compassion, love, and understanding.*

-- Louis Gerstner

Empirical evaluation in software engineering has been gaining importance for many years. It has been recognized (e.g. by [Bas86]) as something that could bring software engineering closer to the established standards of other engineering disciplines. In the last few years empirical evaluations in software engineering have made significant progress by for example integrating practices from the social sciences. However, the evaluation of software development approaches in general is still a difficult (and perhaps hence also a widely disregarded) undertaking that requires a very high degree of effort. Reuse-based approaches are even more problematic in this context since it is by no means a trivial issue to assess the quality of software retrieval approaches as we will explain in the first part of this chapter. However, it is not only the effectiveness of retrieval techniques that influence the practical usability of reuse approaches, but also the tools, the content of the repository, the domain of application, and the underlying process, for example. In other words, a lot of variables may influence the results and it is not always easy to control them. Thus, basic and innovation-oriented research is certainly still justified and necessary. This aligns with the argumentation of [Nun90] who recommend a multi-level evaluation approach to information systems research which we adopted for this thesis as explained in section 1.3. Hence, it was not possible to perform full empirical evaluations on the developed technology at the present time. Instead we evaluate the approach by demonstrating its practical feasibility in smaller proof of concept experiments and small case studies as presented below.

7.1 EVALUATION APPROACHES SO FAR

[Bae99] present two common criticisms of information retrieval (IR) research, namely the lack of a solid formal framework and the lack of consistent testbeds and evaluation frameworks. software engineering in general, and component retrieval in particular, are obviously open to the same criticism. The authors argue that the first criticism is hard to address due to the inherent psychological subjectiveness associated with information understanding by humans. Thus, only the second problem can currently be acted

upon. Retrieval approaches for textual information retrieval are typically compared on so-called reference collections where queries are applied to a well-known collection of documents and the expected results are determined by experts. However, until the so-called TREC (for Text REtrieval Conference) collection with more than one million documents was established in the early 1990s, experimentation in information retrieval had only used small and proprietary “proof-of-concept” test collections for nearly thirty years. However, for the determination of the relevant result a trick had to be applied since the collection is simply too large to be known completely by humans. The reference queries were created by experts and the list of relevant documents was actually created by selecting only the documents that were actually regarded as being relevant by the experts from the results delivered by various IR systems. With this (imperfect) information it has become fairly simple to compare various information retrieval approaches with one another and to calculate recall and precision for them in a comparable way.

Research in software reuse, however, is still many years behind. In the first place, the notion of relevance is typically different compared to textual retrieval systems. While the latter focus on finding potentially meaningful documents in natural language, the basis for component retrieval are typically programming languages and their more formalized constructs. Thus, it is possible to define a much tighter definition of relevance in the software reuse context. In the optimal case, a component is relevant if it matches the required syntactical as well as the semantic properties to 100% and thus can directly be re-used in the given context without any modification. While syntactic matching is essentially a question of pattern matching, it is not guaranteed that a syntactic match also delivers relevant results in terms of functionality. Relevance in textual information retrieval does not require an exact syntactic match, however, as there exist various ways to express the same information with natural language. Actually, this fact is valid for components as well, but a component will only be relevant to a developer if it adheres to the interface defined by him. Thus, we can extend our notion of relevance to all components that deliver the required functionality (i.e. match semantically) and can be adapted to the required interface automatically.

Furthermore, in the component retrieval literature, there is nothing like a common reference collection which would allow component repositories or component discovery algorithms to be evaluated. The few evaluations known so far are all based on proprietary collections with merely a few hundred components ([Fra94], [Pod93]). To date, only [Ino05] has experimented on a significantly larger component base. Furthermore, all three named experiments only applied one retrieval technique. Thus, they were limited to rather imprecise queries and were not able to provide a comparison of different techniques per se. In addition, most previous evaluation attempts suffer from serious methodological flaws that make it difficult to transfer the results to today's conditions. The relevance criterion used from the [Ino05] article, for instance, is not made explicit, but it is likely that it was merely the appearance of a specific term in the source code. However, the fact that we can only assume this, together with the fact that they used a proprietary repository, make it very difficult if not impossible to replicate their study. Other experiments such as those performed by Ye in his Ph.D. thesis [Ye01] to demonstrate the usefulness of his CodeBroker system suffer from additional problems. Due to the small number of components indexed in his prototype, his experimental tasks look very much as if they were optimized for the contents of his repository. Thus, it is very difficult to judge whether his tool would have received such impressive appraisals in a scaled-up version in a productive environment.

7.2 PROOF OF CONCEPT

The background for our evaluation is different, however. We used the open web in our early investigations and now possess a repository with millions of entries that we obviously can no longer oversee manually. Moreover, older retrieval techniques are not precise enough to be used in this context. Thus, as explained before, we combine three retrieval techniques to create our hybrid Extreme Harvesting approach. Since we are not aware of any other similar work at the moment, we cannot directly compare our results with other systems. Thus, the only reasonable approach for evaluating whether our approach is at least equivalent to other approaches is to demonstrate its applicability based on retrieval examples collected from the literature.

Another insight into the demand for component searches and thus a good source for retrieval experiments is provided by Koders.com. Like Google’s “Zeitgeist” search statistics, Koders has started to publish statistics about the most requested search terms for specific programming languages. For example, one popular request to Koders in May 2005 was for an algorithm to calculate the MD5 hash-value for a given string. The following table gives a first impression of the capability of an early Extreme Harvesting prototype using three publicly available search engines as repositories. It shows that it is able to handle the examples of older approaches effectively. The table presents results for various stateless components (i.e. just operations) that contain frequently used algorithms. The first column presents the method names that we used for the search, the second the signature that we entered into our system, the third the number of results discovered on the web from the given search engine in each case, and the last the literature source that provided the inspiration for the example.

Name	Signature	Koders	Yahoo	Google	Source
getRandomNumber	int x int: int	3	6	2	[YeF05], [Ino05]
sort	int[]: void	1	12	15	Koders
reverseArray	int[]: void	0	10	6	-
copyFile	String: void	2	1	0	Koders
isPrime	int: boolean	1	8	14	[Hal93]
sqrt	double: double	2	9	5	[Pod93]
isLeapYear	int: boolean	1	29	24	[YeF05]
randomString	int: String	1	1	0	Koders
replace	String x String: String	14	10	22	Koders
gcd	int x int: int	3	68	10	[Cor01]
md5	String: String	3	1	0	Koders
quicksort	String[]: void	4	3	2	[Ino05]

Table 7.1: Exemplary query results from June and July 2005.

Due to the heuristics implemented in our first prototype, results with slightly different names were adapted to the original signature and also accepted, like `getRandomInt` instead of `getRandomNumber` and so on. However, no parameter permutation was available at that time and hence only results according to orderings that we guessed could successfully be tested. This first proof of concept was based on publicly available search engines accessible over the web. Only one of them (Koders) is a so-called vertical search engine focussing on source code searches. For the other two (Yahoo and Google) we used

the general search interface with the heuristics described in section 4.2. We were actually able to retrieve useful components from them, although [Yao04] still stated at that time that this would be impossible to do. Additionally, these results gave a first indication that our harvesting approach used with general search engines (and the web) at that time performed at least as good as, if not better than the repositories discussed in the literature, and thus encouraged us to pursue our ideas.

We also used this prototype to experiment with more complex and typically stateful components as shown in the following table. Interestingly, we were not able to retrieve a single functioning web service for any of the examples from table 7.1 above and could only find a single web service for the `CreditCardValidator` example from table 7.2 presenting some more complex examples below. This time, for the sake of clarity, we describe the interfaces in the form of UML class diagrams:

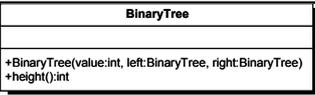
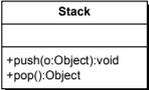
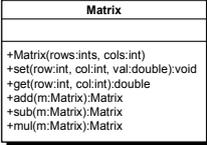
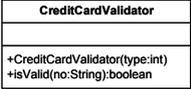
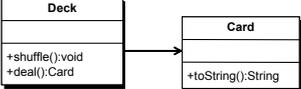
Component's UML diagram	Koders (07/05)	Yahoo (07/05)	Google (07/05)	Merobase (11/07)	Source
	0	4	7	0	[Mil98]
	6	13	33	150	[Ino05] & similar to [Zar95]
	1	1	3	2	[Cza00]
	1	1	1	0	[Vit03]
	0	20	17	20	[YeF05]

Table 7.2: Exemplary query examples for more complex components.

The results demonstrate that even classes and small class ensembles are delivered by our approach with perfect precision (as long as meaningful test cases are supplied, of course). However, it is also apparent that anticipating the correct interface for more complex components rapidly becomes a game of chance and the recall decreases quickly. By way of comparison with the other search engines and our early prototype, we added our latest version of Merobase (with the simple interface-based harvesting) to this

table. Further examples used to mitigate the issue of the decreasing recall and to demonstrate the feasibility of our signature-based, so-called “full” harvesting approach, are discussed in the next section. The test cases used to evaluate the more complex examples in table 7.2 above and in the tables 7.3, 7.6 and 7.8 are listed in appendix A.

7.3 SEMANTIC RETRIEVAL

Earlier, we defined semantic component retrieval as the retrieval of assets the developer really needs and have found test cases are a good vehicle to describe the required functionality. In order to achieve fast retrieval times we use a number of heuristics to cut down the number of candidates to be tested with our retrieval prototype. However, since we have demonstrated the feasibility of a pure test-driven retrieval solution, which is totally independent of any (class or method) names, we want to introduce some examples that illustrate how such a pure test-driven retrieval approach can be used to increase the recall, without losing precision, as long as enough processing power is available. Thus, we have extended our search engine to test all source classes from Merobase's index that contain the required signature. The permutator solution we explained in section 5.4.3 is able to work through all feasible mappings of the required methods to the matching methods in the candidates. For example, consider again a `Stack` data structure which is expected to have the interface shown on the left hand side below. In the pure test-driven retrieval approach, the `Queue` on the right hand side would be a valid candidate (since it is possible to match each method in the `Stack` to a method in the `Queue`) and thus needs to be tested as part of the search process:

```
Stack (
    push(Object) void
    pop():Object
)
Queue (
    enqueueFirst(Object):void
    dequeue():Object
)
```

The above example is not that impressive at a first glance, since a signature-based mapping would be totally sufficient in the cut down version shown. However, the `Queue` candidates found contain a number of other methods such as `enqueueLast(Object)` or `peek():Object` which have the same signature as the required one. Another, more impressive example is the `Calculator` class shown in the table below. Consider its `sub` method which has two parameters, namely the minuend and the subtrahend. The permutator enables our system to consider methods in which the two parameters are contained in reverse order as well. The following table contains some further examples where this approach works well. We present the specified interface in the first column. Columns two and three compare the simple interface-based harvesting with the full signature-based harvesting. We show the number of components that passed the test vs. the total number of candidates in each cell, e.g. for the interface-based harvesting of the `stack`, 150 components passed the test out of 692 candidates. The small numbers show how many candidates passed the test out of the first 10 or 100 etc.

Query	Interface-Based	Signature-Based	Exemplary Result Classes for Signature-Based Harvesting
Stack(push(Object):void pop():Object)	150 /692 2 /10 37 /100 26 min 45 s	611 /35,634 0 /10 1 /100 5 /1000 18 h 23 min	Stack, MyStack, ObjectStack, Queue, Deque, List, LinkedList, Keller, LIFO, Pila, ObjectPool, LifoSet, CircularList
Calculator(sub(int,int):int add(int,int):int mult(int,int):int div(int,int):int)	1 /4 19 s	22 /23,759 0 /100 20 h 24 min	Calculator, CalculatorImpl, Moclecule, Arithmetic, SimpleMath, Operators
Matrix (Matrix(int, int) get(int,int): double set(int,int, double):void multiply(Matrix): Matrix)	2 /10 23 s	26 /137 2 /10 20 /100 5 min 25 s	Matrix
ShoppingCart (getItemCount():int getBalance():double addItem(Product):void empty():void removeItem(Product):void)	4 /4 26 s	4 /12 2/10 47 s	ShoppingCart
Spreadsheet (put(String,String):void get(String):String)	0 /0 3 s	4 /22,705 0 /1000 15 h 13 min	Sheet, Compiler, Util
ComplexNumber (ComplexNumber(double,double) add(ComplexNumber):ComplexNumber getRealPart():double getImaginaryPart():double)	0 /1 3 s	32 /89 1 /10 1 min 19 s	ComplexNumber

Query	Interface-Based	Signature-Based	Exemplary Result Classes for Signature-Based Harvesting
<pre>MortgageCalculator (setRate(double):void setPrincipal(double):void setYears(int):void getMontlyPayment():double)</pre>	<p>0 / 0</p> <p>4 s</p>	<p>15 / 4,265 0 / 100 14 / 1000</p> <p>3 h 19 min</p>	<p>Loan, LoanCalculator, Mortgage</p>

Table 7.3: Comparison of interface-based and signature-based harvesting.

Since our current Lucene index structure delivers a candidate as soon as a required signature appears only once, it is likely that there are actually far fewer classes in our index that contain the signature `int x int -> int` four times as required by the `Calculator` example. Another challenge is highlighted by the `Matrix` example. Since `Matrix` appears as a parameter and return value in the required signature, our current index structure is not able to deliver candidates that have a different name. The increase in results is due to the fact that we have ignored the method names. See section 5.1.2 for a more detailed discussion on this issue. The `ShoppingCart` example demonstrates that our system is also capable of testing classes that depend on other classes. However, at the time of writing, parameter permutation could not be applied to these additional classes. Furthermore, the dependency on the `Product` class makes to the chance of finding similar classes based on the signature rather small since our current implementation requires an exact keyword match.

7.4 PRECISION ANALYSIS

As we pointed out in the section on information retrieval (cf. page 53), retrieval systems are typically evaluated by assessing their precision (proportion of relevant documents amongst the returned documents) and recall (proportion of returned relevant documents). These two measures are often related approximately inversely proportional to one another. The higher the recall the lower the precision and vice versa. Unfortunately, they both require a good knowledge of the underlying document collection to determine their values. More specifically, to calculate the recall an experimenter needs to know the number of relevant documents for a query and to calculate the precision he needs to be able to judge whether a retrieved document is relevant. This issue is normally solved in information retrieval by the use of reference collections as we explained at the beginning of this chapter.

This approach works quite nicely as long as the size of document collections remains in the order of a few thousand. However, the web has allowed search engines to index billions of documents so that no human expert would ever be able to determine all relevant documents for a given query. Thus, calculating the recall is practically impossible for large (web) search engines (cf. [Lew07]). Similarly, it is hard to estimate the precision since it is not known how many relevant results are to be expected. As most users of web search engines, according to empirical investigations, normally investigate only the first 20 results, it makes sense to determine the precision up to a similar reasonable cut off value (this is

also a rationale why a good ranking algorithm is essential for every search engine). This value is called the top-20 precision by [Lew07]. These ideas can be directly transferred to large-scale component search engines. [Ino05] already recognized these problems when they experimented with their ComponentRank approach with about 150,000 components. For our Merobase engine with a total of 10 million entries these problems have become even greater.

The aim of the subsequent subsections, therefore, is to present reasonable evidence for the effectiveness of our component search engine in supporting the use cases introduced in section 5.1. However, since the library searches that we identified as use cases (and textual searches as well) are simply based on keyword matching, which has been a standard technique for many years, we do not invest any time in evaluating these (see e.g. [Gar06] for such an example). We rather evaluate our technology for the other three search algorithms in the next subsections thoroughly, starting with an evaluation of open source searches below.

7.4.1 EVALUATING OPEN SOURCE SEARCHES

The evaluation of open source searches is a relatively simple undertaking compared with the speculative and definitive component searches, which will follow later. Remember, open source searches are supposed to deliver the source code of a specific class from a specific open source project with as little effort as possible for the searcher. According to our opinion, the “projectname classname” constraint is the most appropriate way to submit this information to a search engine. We discussed the motivation and the implementation of it in more detail on page 87 et seq. Once it is known that a specific open source project (such as Lucene) is in the index, it is straightforward to search for a specific class within it (e.g. QueryParser) and to determine whether the original version is contained in the top-10 results, for example. This approach can be seen as a variant of the so-called “known item” test defined by [Kan76]. It has been used in its original form to evaluate the performance of libraries in delivering known books to a customer.

The following table summarizes the results of the experiment which we performed to evaluate the precision in retrieving open source entities. We compared our optimized Merobase algorithm, with the regular Lucene algorithm and a name-based Latent Semantic Indexing [Dee90] algorithm as described in [Gru07] on Merobase. Furthermore, we added Koders and Krugle, two other search engines that claim to index the most important open source repositories to our comparison as well.

Query	Optimized (Merobase)	Keyword (Merobase)	LSI (Merobase)	Koders	Krugle
ant junittask	1 / 236	x / 47	x / ∞	x / 25	1 / 667
eclipse astparser	1 / 484	x / 360	x / ∞	x / 100	2 / 660
eclipse navigator	1 / 2,727	x / 2,562	x / ∞	x / 1,027	1 / 10
eclipse textelement	1 / 215	x / 147	x / ∞	2 / 19	1 / 342
findbugs redundantbranch	1 / 17	1 / 8	x / ∞	2 / 4	1 / 8

Query	Optimized (Merobase)	Keyword (Merobase)	LSI (Merobase)	Koders	Krugle
jmaki jmakicontroller	1 / 10	1 / 4	1 / ∞	1 / 1	1 / 7
juddi registryobject	1 / 1172	1 / 1027	x / ∞	- / 9	1 / 701
junit testcase	1 / 100,809	x / 100,280	x / ∞	x / 33,616	3 / 121,254
lucene queryparser	1 / 2248	x / 983	x / ∞	x / 530	3 / 1923
opensymphony rijndael	1 / 42	1 / 2	x / ∞	1 / 4	1 / 4
struts submitaction	1 / 86	1 / 25	x / ∞	- / 24	1 / 115
tomcat url	1 / 2,786	x / 2,446	x / ∞	x / 850	1 / 15,561

Table 7.4: Comparison of retrieval performance for open source searches on various search engines.

In front of the slash in each cell we present the position at which the first appropriate result was ranked by each search engine and after the slash we present the total number of delivered results. An “x” means that the correct candidate was not amongst the top 10 results and an “-” indicates that the project was not indexed by this engine at all. For the LSI algorithm we denoted an infinite number of results since LSI calculates the distance of each document to the query and simply delivers the document with the lowest distance first. There is no concrete threshold that determines whether or not a document is relevant.

It is interesting that only the optimized algorithms of Krugle and Merobase, developed independently and roughly at the same time as one another, reliably deliver the desired results amongst the highest ranked candidates. The other algorithms are only able to come up with some chance hits for not particularly well-known projects where the total number of hits is apparently much lower anyway. As soon as a larger number of potential results is to be delivered, their performance totally breaks away. The LSI algorithm is a special case in this context. As we also realized in other experiments, the perceived results of the LSI algorithm in general are not bad (considering what it is supposed to do) since candidates somehow related with the query are usually delivered first. On the other hand, however, the results are too general since the existing more concrete results get lost within the “noise” of somehow reasonable, but not actually relevant results.

7.4.2 COMPARISON OF RETRIEVAL TECHNIQUES

To evaluate the performance of speculative and definitive searches we started with reusing the query examples we had collected from the literature for our proof-of-concept implementation. We performed these interface-driven queries again and inspected the first 25 results for each query to judge whether they offered the functionality we were expecting. Our matching criterion was that either the required signature was completely contained (verbatim) in a candidate or was contained with only a change of case and that the associated JUnit test cases were successfully passed. In other words, we applied our Extreme Harvesting approach as the final criterion for determining whether or not an asset was relevant.

We performed two different experiments. First, we used our Extreme Harvesting prototype to evaluate the retrieval performance of various search engines on the web as shown in table 7.5. We used, to the best of our understanding, the most precise queries for achieving interface-based retrieval for each search engine. We limited our comparison to the three component search engines shown in the table below since only they offered an API for programmatic access. We evaluated how they compared with the general web search versions of Google and Yahoo enhanced with special filetype constraints (as explained in section 4.2) to better retrieve software components. In total, we searched for twelve functional abstractions in the first part of the experiment.

Query	Google	Yahoo	GCS	Koders	Merobase
<code>copyFile(String, String):void</code>	1 / 25	2 / 25	7 / 25	0 / 25	18 / 25
<code>gcd(int,int):int</code>	10 / 25	7 / 25	12 / 25	2 / 25	17 / 25
<code>isLeapYear(int):boolean</code>	8 / 25	12 / 25	3 / 25	2 / 25	14 / 25
<code>md5(String):String</code>	0 / 25	0 / 25	4 / 22	0 / 25	12 / 25
<code>isPrime(int):boolean</code>	6 / 25	15 / 25	7 / 25	4 / 25	5 / 25
<code>randomNumber(int,int):int</code>	0 / 25	3 / 25	2 / 7	0 / 7	14 / 25
<code>randomString(int):String</code>	4 / 25	2 / 25	6 / 25	4 / 16	5 / 25
<code>replace(String, String, String):String</code>	2 / 25	8 / 25	14 / 25	3 / 25	22 / 25
<code>reverseArray(int[]):int[]</code>	1 / 10	3 / 23	1 / 1	0 / 4	5 / 7
<code>sort(int[]):int[]</code>	0 / 25	0 / 25	5 / 20	0 / 25	20 / 25
<code>sqrt(double):double</code>	5 / 25	4 / 25	4 / 25	1 / 25	11 / 25
<code>getMinMax(int[]):int[]</code>	0 / 15	0 / 22	0 / 0	0 / 25	2 / 4
Average Precision	12.8%	18.4%	32.0%	6.1%	56.1%
Standard Deviation	13.7%	19.7%	26.1%	8.1%	21.5%

Table 7.5: Comparison of code search engines performed on stateless operations.

We then calculated the mean value and the standard deviation of each engine's precision. Furthermore, we performed t-tests for $\alpha = 0.05$ to measure the statistical difference of the results. Only the results provided by Merobase show a significant improvement over those of other engines. Google Codesearch (GCS) is also significantly better than Koders, but all other pairwise comparisons reveal no statistically significant difference. It is interesting that the general versions of Google and Yahoo even seem to deliver more precise results for code searches than the specialized engine of Koders. However, we believe that

this can be explained by the different expressiveness of the queries that can be used with the different search engines. We will back this up with more evidence in the next paragraph where we compare interface-driven retrieval with other methods.

For table 7.5 above we merely used simple stateless operations, but to get a better impression of the performance of the engines when dealing with full-fledged objects, we repeated the same experiment with the following collection. However, as we (and the literature) assume the likelihood of correctly guessing the interface of an object decreases with its complexity and thus, not too much significance should be placed on the results since positive matches only occur when the correct interface is found.

Query	Google	Yahoo	GCS	Koders	Merobase
Account (deposit(double):void withdraw(double):void getBalance():double)	1 / 21	7 / 25	8 / 25	0 / 25	6 / 25
Article (setId(int):void setName(String):void setPrice(double):void getId():int getName():String getPrice():double)	0 / 1	0 / 2	0 / 2	0 / 0	4 / 4
Calculator (add(int,int):int subtract(int,int):int mult(int,int):int divide(int,int):int)	0 / 2	1 / 5	0 / 0	0 / 0	1 / 4
ComplexNumber (add(ComplexNumber):ComplexNumber getRealPart():double getImaginaryPart():double)	2 / 25	0 / 3	0 / 2	0 / 1	0 / 1
Customer (setAddress(String):void getAddress():String)	3 / 25	5 / 25	1 / 25	1 / 25	13 / 25
Die (roll():void getFaceValue():int)	7 / 25	7 / 25	3 / 3	0 / 0	2 / 4
Document (Document(String,String,String) getAuthor():String getTitle():String)	0 / 25	0 / 25	0 / 21	0 / 8	0 / 25

Query	Google	Yahoo	GCS	Koders	Merobase
Matrix (Matrix(int, int) set(int, int, double):void get(int, int):double multiply(Matrix):Matrix)	0 / 25	0 / 25	0 / 6	0 / 25	2 / 10
Movie (Movie(String, int) getTitle():String)	1 / 25	3 / 25	1 / 25	2 / 9	15 / 25
Sort (quickSort(int[]):void)	0 / 25	3 / 25	11 / 25	0 / 25	5 / 16
Spreadsheet (put(String, String):void get(String):String)	0 / 22	0 / 25	0 / 0	0 / 0	0 / 0
Stack (push(Object):void pop():Object)	2 / 25	4 / 25	0 / 25	6 / 25	5 / 25
Average Precision	5.4%	11.3%	15.3%	4.2%	31.9%
Standard Deviation	8.2%	11.2%	30.4%	8.9%	29.6%

Table 7.6: Comparison of search engines with small exemplary components.

The results in table 7.6 confirm the intuitive assumption of [Sam97] that the complexity of a component has a significant influence on its precision and recall (at least as long as no sophisticated tool support is available as we demonstrated in section 7.3). For instance, complex components such as Matrix or Spreadsheet are noteworthy since there are no returned candidates. Compared to the operations from table 7.5 the overall precision values drop by nearly 20%. According to a t-test for $\alpha = 0.05$ the difference in the Merobase examples is even statistically significant. The difference between the values in this table is thus not as clear as before. Although the results indicate that Merobase has the highest precision again, the difference is statistically significant only to Koders and mainstream Google. Compared with Yahoo and Google Codesearch the difference is not significant this time.

The second experiment was an academic comparison of the four retrieval techniques and the associated representation methods as introduced in section 3.2.11. Table 7.7 presents the results of our experiment, this time performed completely on the data pool of Merobase with various retrieval techniques. The experimental process is identical to the one summarized in the paragraph above. We compared interface-driven search capabilities with pure signature matching and with simple keyword-based searches in two distinct forms. Namely, our “speculative” algorithm which tries to guess the functionality of a component by placing special emphasis on some keywords and a name-based algorithm which is able to constrain searches to method and class names (similar to the capabilities that Krugle and Koders offer). In addition to the precision for the first 25 values as used before, we also show the total number of results delivered by a retrieval technique italicized in each cell.

Query	signature matching	speculative keyword matching	name-based	interface-driven
<code>copyFile(String, String):void</code>	0 / 25 <i>63,904</i>	3 / 25 <i>3,023</i>	16 / 25 <i>3,305</i>	18 / 25 <i>315</i>
<code>gcd(int, int):int</code>	0 / 25 <i>21,690</i>	20 / 25 <i>1,752</i>	11 / 25 <i>1,998</i>	17 / 25 <i>523</i>
<code>isLeapYear(int):boolean</code>	0 / 25 <i>38,967</i>	9 / 25 <i>467</i>	7 / 25 <i>563</i>	14 / 25 <i>280</i>
<code>md5(String):String</code>	0 / 25 <i>131,281</i>	0 / 25 <i>447</i>	0 / 25 <i>515</i>	12 / 25 <i>55</i>
<code>isPrime(int):boolean</code>	0 / 25 <i>38,967</i>	4 / 25 <i>724</i>	5 / 25 <i>872</i>	5 / 25 <i>357</i>
<code>randomNumber(int, int):int</code>	0 / 25 <i>21,690</i>	0 / 25 <i>553</i>	0 / 25 <i>607</i>	14 / 25 <i>31</i>
<code>randomString(int):String</code>	0 / 25 <i>120,997</i>	4 / 25 <i>370</i>	6 / 25 <i>155</i>	5 / 25 <i>72</i>
<code>replace(String, String, String):String</code>	1 / 25 <i>7,775</i>	6 / 25 <i>81,840</i>	0 / 25 <i>92,385</i>	22 / 25 <i>1473</i>
<code>reverseArray(int[]):int[]</code>	0 / 25 <i>1,848</i>	0 / 25 <i>90</i>	2 / 25 <i>93</i>	5 / 7 <i>7</i>
<code>sort(int[]):int[]</code>	1 / 25 <i>1,848</i>	0 / 25 <i>60,246</i>	0 / 25 <i>67,669</i>	20 / 25 <i>68</i>
<code>sqrt(double):double</code>	0 / 25 <i>12,285</i>	2 / 25 <i>25,430</i>	4 / 25 <i>30,583</i>	11 / 25 <i>258</i>
<code>getMinMax(int[]):int[]</code>	1 / 25 <i>1,848</i>	2 / 25 <i>289</i>	2 / 25 <i>298</i>	2 / 4 <i>4</i>
Average Precision	1.0%	16.7%	17.7%	56.1%
Standard Deviation	1.8%	22.8%	20.1%	21.5 %

Table 7.7: Comparison of retrieval techniques on stateless operations.

We also performed statistical t-tests for $\alpha = 0.05$ on these results and found all pairwise comparisons significantly different, except for speculative vs. name-based. We also ran the same experiments with the more complex components that we used before. These results are shown in the following table.

Query	signature matching	speculative keyword matching	name-based	interface-driven
Account (deposit(double):void withdraw(double):void getBalance():double)	0 / 25 <i>12,245</i>	6 / 25 <i>556</i>	5 / 25 <i>1,104</i>	6 / 25 <i>93</i>

Query	signature matching	speculative keyword matching	name- based	interface- driven
Article (setId(int):void setName(String):void setPrice(double):void getId():int getName():String getPrice():double)	0 / 25 4,166	4 / 5 5	4 / 5 5	4 / 4 4
Calculator (add(int,int):int subtract(int,int):int mult(int,int):int divide(int,int):int)	1 / 25 1,283	1 / 12 12	1 / 12 12	1 / 4 4
ComplexNumber (add(ComplexNumber):ComplexNumber getRealPart():double getImaginaryPart():double)	0 / 25 1,285	0 / 1 1	0 / 1 1	0 / 1 1
Customer (setAddress(String):void getAddress():String)	0 / 25 1,552	6 / 25 410	6 / 25 425	13 / 25 54
Die (roll():void getFaceValue():int)	0 / 25 198,365	19 / 25 63	22 / 25 115	2 / 4 4
Document (Document(String,String,String) getAuthor():String getTitle():String)	0 / 25 3,892	0 / 25 332	0 / 25 337	0 / 25 25
Matrix (Matrix(int,int) set(int,int,double):void get(int,int):double multiply(Matrix):Matrix)	0 / 25 73	0 / 25 551	0 / 25 803	2 / 10 10
Movie (Movie(String,int) getTitle():String)	0 / 25 11,396	12 / 25 264	10 / 25 318	15 / 25 29
Sort (quickSort(int[]):void)	0 / 25 11,826	0 / 25 2,692	0 / 25 2,824	5 / 16 16
Spreadsheet (put(String,String):void get(String):String)	0 / 25 22,153	0 / 25 244	0 / 25 295	0 / 0 0

Query	signature matching	speculative keyword matching	name- based	interface- driven
Stack (push(Object):void pop():Object)	0 / 25 <i>33,844</i>	2 / 25 <i>11,505</i>	0 / 25 <i>20,641</i>	5 / 25 <i>692</i>
Average Precision	0.3%	22.4%	21.7%	31.9%
Standard Deviation	1.2%	29.8%	31.8%	29.6%

Table 7.8: Comparison of retrieval techniques.

The differences in this table are not significant for $\alpha = 0.05$ between the last three columns (i.e. speculative, name-based and interface-based). However, they all are significantly different from the results for signature matching in the first column. The results for the `Die` example are of particular interest – indeed, they are amazing at a first glance. The interface-based searches deliver only four potentially matching candidates while a name-based search delivers 22 positively tested candidates that should have the same interface, of course. The answer for this little paradox is simple. Only a few of the results have a `roll` method that actually returns `void`. Most of the candidates return an `int`, which is of course recognized and excluded by the interface-based search algorithm, but is accepted by the Java compiler and ignored by the test case.

When combined, the above comparisons of retrieval techniques demonstrate that interface-driven searches are usually better than plain keyword-based queries in terms of suggesting candidates that are likely to also be semantically appropriate. However, the latter tend to lose some of their advantage when components become more complex, which is, of course, understandable since the more information an interface contains, the more descriptive text can be used for the weaker retrieval techniques and the less false positives they will return. Furthermore, these results also explain why Koders tends to be even weaker than the general versions of Google and Yahoo where interface-driven searches can be simulated to a certain extent. Koders merely implements a very simple text-based algorithm that could not compete with the more sophisticated approaches of the other engines.

However, despite the promising results shown in these experiments, the overall precision values remain roughly between 30 and 60 percent and given the fact that sometimes thousands of useless candidates are returned a further increase in the precision is urgently required. This is another clear hint to use a final semantic assessment of the candidates as integrated in our Extreme Harvesting approach. However, another important requirement for precise searches in the practical use of large-scale component search engines are so-called search constraints that allow queries to be constrained to a given language or component type as is common in most web search engines today. Otherwise, the results in the above experiments would certainly have been worse.

To summarize, the results in this subsection support three of the assumptions that we postulated earlier on. First of all, they underline that we are actually on the right track by combining various retrieval techniques from the literature to increase the precision of the overall retrieval approach used on today's

large component collections. Second, it is also important to incorporate not only one, but a number of representation methods (as introduced in section 3.1.1) to focus the retrieval algorithms on the right programming language or component type, for instance. And third, the experiments have shown that the more complex a component becomes the lower the chances for a “lucky guess” of its interface become and the greater the need for an approach with increased recall such as that based on signature-based harvesting as introduced in section 7.3.

7.5 CASE STUDY

Finding a good case study for a reuse system such as our Merobase Eclipse plugin is another problematic task since on the one hand it has to be general enough to contain some reusable elements, and on the other hand it needs to be specific enough to go beyond the level of simple data structures. Since we want to demonstrate our approach embedded in a test-driven development context, we studied popular literature in this area. However, this was not very helpful since some books such as [Fow99] are so popular that their code examples can be found on the web hundredfold. Other books such as [Bec03] only contain extremely simple examples, or examples that are so specific they never made their way on the web as in [Bec99]. Luckily, we finally found the book by [Wak02] that develops a simple search system for bibliographic data as a running example. The following four classes comprise its initial “design”.

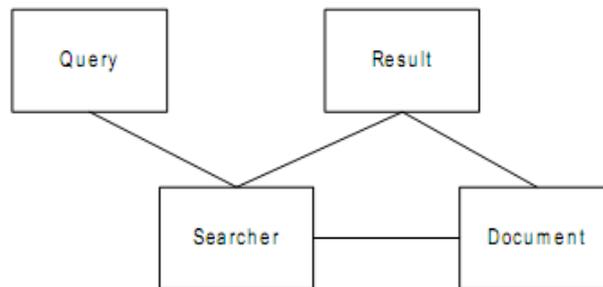


Figure 7.1: Initial design of Wake's running example.

In the following, the system is developed in a fully test-driven manner. In other words, Wake starts with the definition of tests and derives the interfaces of the classes from them. Since all other classes depend on the `Document` class, it makes sense to define the following simple unit test for the `Document` class first.

```

public class DocumentTest extends TestCase {
    public void testDocument() {
        Document d = new Document("a", "t", "y");
        assertEquals("a", d.getAuthor());
        assertEquals("t", d.getTitle());
        assertEquals("y", d.getYear());
    }
}

```

In a classic TDD approach a developer would then create a `Document` class, add an empty constructor with three `String` parameters and the three empty getter methods. Our Merobase plugin makes this step much easier. As soon as we have created the test case shown above in a project, it suggests two `Document` classes that are likely to comply with the unit test, at least syntactically. And a few moments later, it will have tested them (remotely on the secured server environment) and found that the first result also passes the test as shown in the following picture:

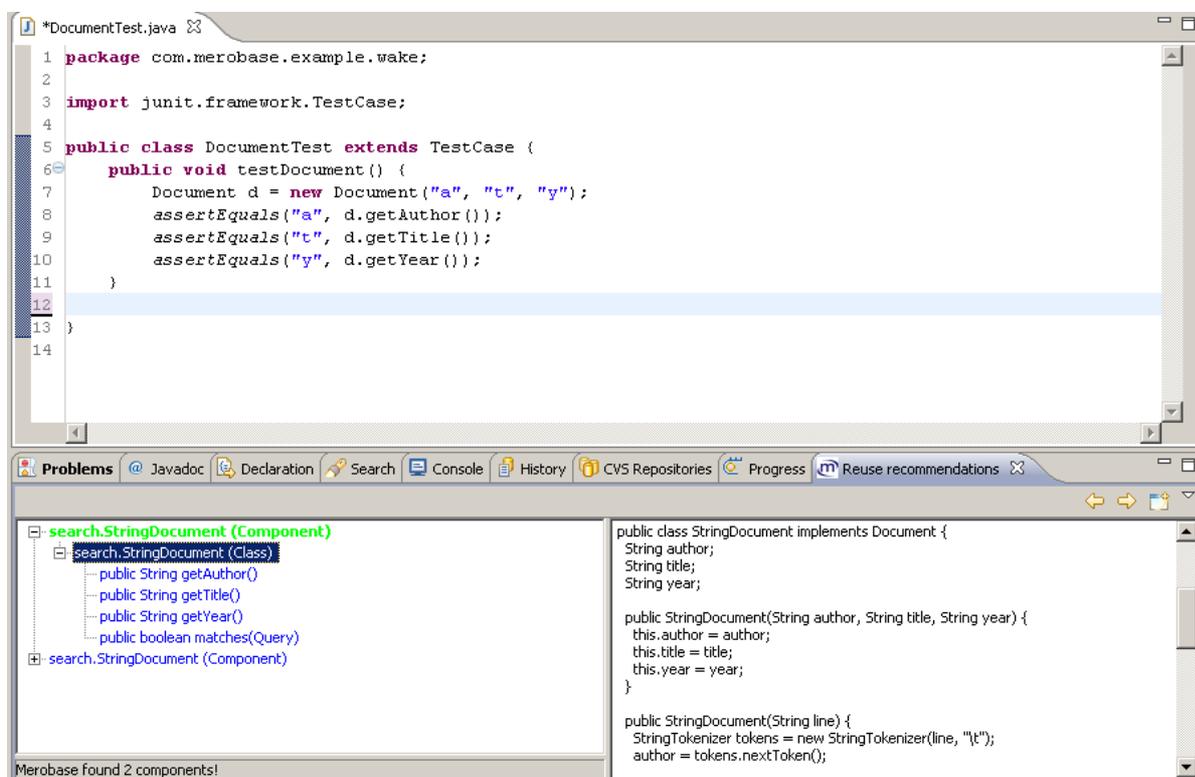


Figure 7.2: Screenshot of the Eclipse plugin recommending a `Document` class.

A double click on the `StringDocument` class copies its source code into the current project and after renaming it to `Document`, removing the superfluous `Document` interface and commenting out a method that relies on the `Query` class, our candidate is properly integrated into our project. In the future, these steps should happen automatically, of course. The next step is executing our test case locally on the candidate to ensure that we have integrated it correctly, which will lead to a green test result from JUnit.

Wake then starts to define the semantics of the `Result` class with the following test case, which returns a total of twelve syntactically matching candidates, one of which actually passes the test.

```
public class ResultTest extends TestCase {
    public void testEmptyResult() {
        Result r = new Result();
        assertEquals (0, r.getCount());
    }
}
```

We repeat the process from above by double clicking on the result to integrate it into the project, performing some minor manual adaptations and executing the unit test locally. Since this is passed without problem, we can continue by adding the next unit test proposed by Wake for the `ResultTest` class:

```
public void testResultWithTwoDocuments() {
    Document d1 = new Document("a1", "t1", "y1");
    Document d2 = new Document("a2", "t2", "y2");
    Result r = new Result(new Document[]{d1, d2});
    assertEquals (2, r.getCount());
    assertTrue(r.getItem(0) == d1);
    assertTrue(r.getItem(1) == d2);
}
```

To make sure the newly delivered `Result` and the `Document` classes from before are correctly integrated, we execute the test case once more and again receive a green bar from JUnit. We are now ready to continue with the `Query` class by starting with a test case as follows:

```
public class QueryTest extends TestCase {
    public void testSimpleQuery() {
        Query q = new Query("test");
        assertEquals("test", q.getValue());
    }
}
```

This results in 5 syntactically matching candidates proposed by our plugin, one of which passes the test. We integrate it into our project as before and run the test case locally to ensure it actually works as expected. The `Searcher` class is the next to be considered. Wake defines the following test case for it.

```
public class SearcherTest extends TestCase {
    public void testEmptyCollection() {
        Searcher searcher = new Searcher();
        Result r = searcher.find(new Query("any"));
        assertEquals(0, r.getCount());
    }
}
```

Unfortunately, this time, the compilation service on the Merobase server is not able to resolve the various dependencies contained in the four `Searcher` candidates that match syntactically and thus we have to inspect these classes manually to assess whether or not we can use one of them. In fact, one of the classes can be adapted with about five or six minor changes so that we finally receive a green bar for this test case. At this point we have harvested an initial version of all four classes required for Wake's system.

Wake's example implementation in the book contains only the absolute minimum code that enables it to pass the unit tests. However, the code harvested in our case study obviously is not some intermediate implementation, but a more sophisticated version. Consequently, it should be able to pass the additional test cases that Wake creates to cover the rest of his implementation. We added them to the appropriate test classes and executed them to see whether all functionality was implemented correctly by the harvested files. After re-adding the method that was commented out earlier in the `Document` class and including two constructors that were commented out in the downloaded version of the `Document` and the `Result` class, we were in fact able to run all JUnit test cases successfully as shown in the following figure:

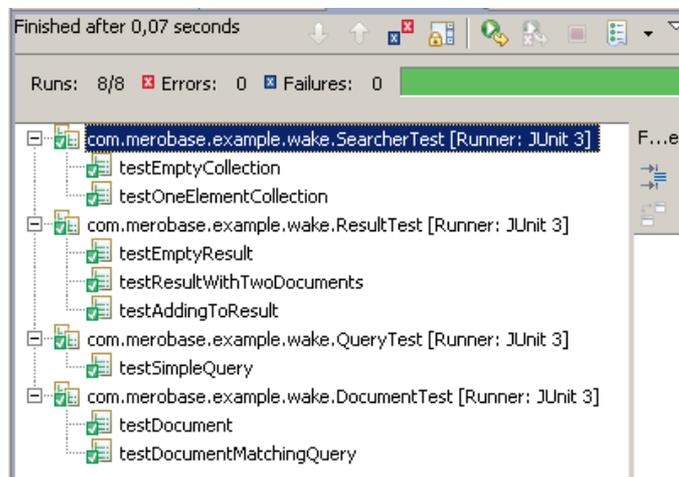


Figure 7.3: Screenshot of the final local JUnit test run.

In total, we needed only about ten minutes to find all four classes that fully implemented the functionality defined in Wake's test cases, while programming it would have certainly taken more than an hour. Thus, although this small case study is somewhat artificial, it demonstrates the high potential of the approach and suggests that further analysis of its usability in industrial environment are warranted.

8 RELATED WORK

If I have seen further it is by standing on the shoulders of giants.
-- Isaac Newton

In earlier parts of this thesis we introduced the seminal retrieval techniques that paved the ground for our work, and described other relevant research projects and prototypes that have been developed over the last decade. We reported on all related work that is fundamental to the understanding of this dissertation or that was used for the purposes of comparison. In this chapter we summarize the latest developments in the software reuse community with a special focus on working systems that aim to component-based software reuse. The study of the advantages and disadvantages of these systems provided many valuable insights used in the development of Extreme Harvesting and the associated prototype. We are currently not aware of any system that has the same capabilities as that developed for this dissertation: Extreme Harvesting is to our knowledge the only practically implemented component retrieval approach that offers support for queries with linguistic and syntactic filtering steps and a semantic assessment that assures a high degree of component suitability, i.e. precision. On the other hand, the idea of using the web as a source for reusable components, as documented in the next subsection, was not new when we started our research. However, as we shall also see, where attempted it has not been developed to a practically usable technology.

8.1 COMPONENT SEARCH ON THE INTERNET

As mentioned earlier, component search on the Internet was neglected by researchers for quite a long time. Even in 2004, some researchers still publicly stated that a web search for components would not be feasible at all [Yao04]. Hence, we are currently only aware of one research project that has tried to discover components from the web with the help of a search engine. And only recently have some commercial approaches came up trying to utilize the large amounts of source code available on the Internet. We will present a brief outline of these developments in this section. We begin by providing a more detailed discussion of the so-called Agora system, already mentioned a few times before.

8.1.1 AGORA

Robert Seacord and his colleagues at the Software Engineering Institute (SEI) were the first researchers to publish substantial work on utilizing the web as a source for software components. Already back in 1998, they published their work about Agora, a system that was designed to use introspection mechanisms for identifying and retrieving JavaBeans¹² and CORBA components from the web. At that time the AltaVista.com search engine distributed the AltaVista Search Developer's Kit, which offered search engine capabilities like indexing and searching with a C++ API. Seacord and colleagues used this to set up their own component repository filled by agents that crawled the web for specific component types. The agents typically searched the web with the help of the AltaVista search engine for their respective component type. If they discovered a suitable component, they used mechanisms such as Java introspection to extract interface information from the component and sent it to the index server that stored it in the index. This index was connected to a so-called query server, which in turn was connected to a web server that was responsible for forwarding the user's queries to the query server and returning results to the user's web browser. We would probably call this a 3-tier architecture today. Figure 8.1 depicts it graphically.

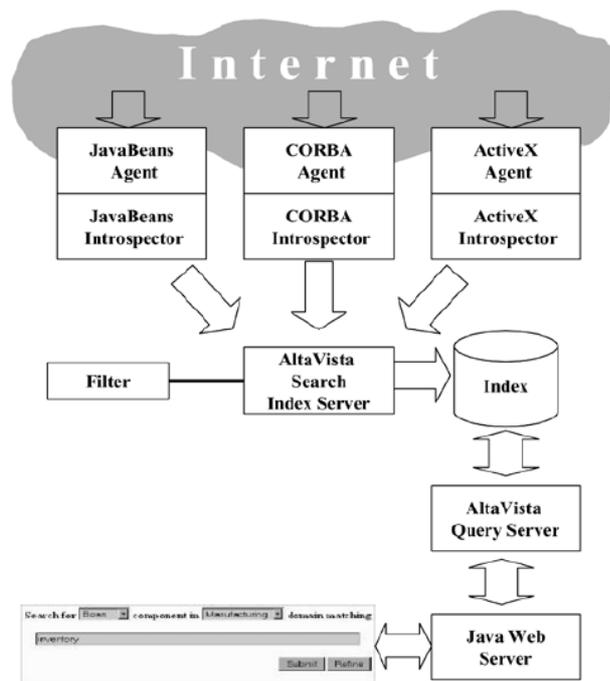


Figure 8.1: Agora's architecture, taken from [Sea98].

The figure shows three kinds of components that were planned for the system, namely JavaBeans, CORBA and ActiveX. In practice, however, significant success could only be achieved for JavaBeans. According to [Sea98], the retrieval of CORBA components raised a lot of problems and the developers

¹²The authors carelessly intermingle the terms JavaBeans and Java applets in their article and so far as we are able to make out only worked with Java applets that are embedded in web pages.

ended up recommending that the OMG adapt CORBA significantly to make it suitable for Agora. Similarly, the ActiveX agent never made it beyond the prototype stage.

The major drawback of the Agora idea (at least at that time) was the additional layer of complexity of the index server that the developers imposed on their system. As they reported, filling the index server with components was a major performance bottleneck and allowed only about 800 JavaBeans to be indexed in 24 hours, even with a cluster of crawlers. Given that around 150,000 Java applets that were available on the web at that time through the AltaVista search engine, it would have taken more than half a year to add all these applets to Agora's index. Unfortunately, the authors did not reveal how many applets they had actually indexed, but some other numbers they presented indicate that it was in the order of just a few thousand. Thus, the Agora experiment has to be regarded as a failure due to the ambitious architecture that could not be implemented with the hard- and software resources available at that time.

8.1.2 WEB-BASED COMPONENT & CODE SEARCH

About ten years after Agora, the high potential for - component retrieval from the Internet, and thus of this dissertation, was underlined by the fact that a large number of component search engines emerged during the time that this dissertation work was performed. These various engines all reinforced the belief that the repository problem could now be regarded as solved since they all collected a significant number of components, sometimes in the millions. Details on the size of the most important search engines has already been presented in section 4.1.1. However, most of them are still limited to rather primitive keyword-based query algorithms and thus demonstrated that the retrieval problem still has not been solved in general. Furthermore, since most of these engines are commercial, there is only minimal information about their internal structure and implementation. However, since the research of this thesis has triggered the development of Merobase, one of the most advanced component search engines available today, we refer the reader to section 4.3 for a more detailed description of how this can be done.

In addition to our Merobase search engine and the three other important search engines already discussed in section 4.1.1, a large number of other code search engines has appeared on the world wide web in the last two or three years. We surveyed them in summer 2007 and give a brief summary of their most important features in the following table. Since interface-based driven and signature-driven searches are obviously not that simple to support, some of the search engines introduced a new retrieval technique allowing certain constraints to be defined on class and method names. In chapter 7, we have called this *name-based* searches and also assessed their capability in comparison to the other retrieval techniques.

URL	Languages	Size	Java Files	Sources	Search Types	Constraints
Merobase.com	5 (+ 43 by GCS)	>10 M	8 M	CVS SVN HTTP	1) keyword 2) name-based 3) sign. matching 4) interface-based 5) spec.-based	form, type, kind, namespace, project, url, host, license, lictype, name, method
krugle.com	43	>10 M	3.5 M	CVS	1) keyword	language, project,

URL	Languages	Size	Java Files	Sources	Search Types	Constraints
				SVN	2) name-based	filename, site, classdef, functiondef, functioncall, comment, code, file extensions
google.com/codesearch	46	> 10 M	2.5 M	CVS SVN HTTP	1) keyword 2) regex	lang, file, package, license
koders.com	32	> 1 M	600 k	CVS SVN	1) keyword 2) name-based	lang, licence, cdef, mdef, idef, file
codase.com	3	< 1 M	300 k	CVS	1) keyword 2) name-based 3) interface-based	lang, project
codefetch.com	22	< 100 k	< 100 k	Books	1) keyword	lang
csourcsearch.net	2	1 M	0	CVS	1) keyword 2) name-based	various code elements
labs.oreilly.com/code	25	100 k	15 k	Books	1) keyword	cat, isbn, author, pubyear, chapter
ucodit.com	2	> 100 k	> 100 k	SVN	1) keyword 2) name-based	-
mine8.ics.uci.edu: 8080/sourcerer2/search/index.jsp	1	250 k	250 k	CVS	1) keyword 2) topological	comments
demo.spars.info	2	> 300 k	300 k	CVS	1) keyword 2) name-based	-
planetsourcecode.com	11	< 100 k	< 50 k	uploaded	1) keyword	lang, category, code type, code difficulty level
yahoo.com (originfileextension:java)	all	> 10 M	> 500 k	HTTP	1) keyword 2) name-based	url, site, title
componentsource.com	8	> 1000	> 100	proprietary	1) keyword	-

Table 8.1: Overview of recent code search engines.

As the table illustrates, the vast majority of code search engines available today are still limited to primitive keyword matching retrieval techniques. Furthermore, most of them are limited to components from CVS or SVN repositories, with only Merobase and Google Codesearch including content from version control repositories and the open web in their indices. We have also listed the regular versions of Google and Yahoo as well as the component broker componentsource.com to facilitate comparison. It is interesting to mention that – according to our experiments in section 7 – the two general style search engines have a higher potential in terms of precision than most specialized search engines if their undocumented features are well exploited.

In addition to the search engines just presented, there are some other even more specialized search engines available on the web focussing on the so-called library searches we introduced in section 5.1.3. For the sake of completeness, we present an overview of them and a comparison with Merobase in the subsequent table.

URL	JAR Files	Class Files
Merobase.com	n.a.	4 M
jarhoo.com	10 k	500 k
jarfinder.com	n.a.	250 k
whatjar.net	not functioning as of 11/2007	

Table 8.2: Overview of search engines offering library searches.

As the table demonstrates our collection in Merobase is the largest collection of class files retrieved from JAR libraries and is about eight times larger than the closest competitor.

8.1.3 WEB 2.0 TECHNOLOGIES

The recent Web 2.0 hype has also left its mark on software reuse. However, although Web 2.0 was originally regarded as the precursor to the Semantic Web [Ber01], the term Web 2.0 in widespread use today goes back to a definition by Tim O'Reilly [ORe05] and merely covers techniques that allow users to influence the content available on the web, e.g. wikis, blogs, tagging etc. However, the idea that users should be able to edit web pages (similar to what we would call a wiki today) was already contained in Tim Berners Lee's original vision for the "Web 1.0" [Ber99]. Nevertheless, community-driven websites based on the ideas of today's Web 2.0 have become widely available (e.g. del.icio.us etc.).

The emergence of similar community sites could be observed in the software engineering community recently (such as planetsource.com). Sites that offer users the possibility to upload, comment and sometimes tag source code or source code snippets can be considered as collaborative reuse or at least knowledge sharing platforms. Even some of the main code search engines (such as Krugle) are offering commenting capabilities at the time of writing. However, currently there is few, if any, scientific data on how useful these "gimmicks" really are in the context of component search. We are only aware of one paper where tagging was investigated as an additional data layer for component searches [Van07]. Interestingly, the authors describe that the high upfront effort involved in adding tags to their small database of a few hundred classes did not lead to any significant improvements in search quality. Rather, the search results were worse than those achieved with simple keyword-based searches. Since source code (or components) contain much more structural information than plain web pages, we assume that it is highly questionable whether techniques like tagging are applicable in this context. Perhaps, tagging might be useful for component management in closed developer teams, but for open repositories we expect tagging to experience the same problems as other linguistic approaches (see e.g. [Fur87]). The only thing we can say for sure at this time is that Web 2.0 technologies for component reuse still require a lot of research to determine whether they will be able to increase component search efficiency.

8.2 OTHER REUSE TOOLS

Since the idea of software reuse has been around for almost four decades there have been many attempts to create viable reuse technologies including the implementation of various reuse tools. The number of such tools reported in the literature so date is probably in three digits. As a result, there is almost no idea related to software reuse that has not been tried out in one tool or another. It is clearly too much to discuss each of these tools in the scope of this thesis but below we discuss the ones whose ideas have been most influential for this thesis and simultaneously hope to provide a reasonable overview of the state of the art. Unfortunately, most of these tools are not available for testing today, as since they either are not working with today technology, require complicated configuration or even worse are just not available any more. As a consequence, it is only possible to summarize information from the literature and not to compare these tools in action.

8.2.1 CODEBROKER

The CodeBroker system was developed by Yunwen Ye in his Ph.D. dissertation [Ye01] at the University of Colorado. It is certainly one of the most influential and interesting works concerned with software reuse in recent years and is documented by the numerous papers that have been published about it since the year 2000. Just recently a summary article [YeF05] appeared in the Journal of Automated Software Engineering. Admittedly, the main objective of Ye's work was not to develop a new reuse or retrieval mechanism, but rather to explore a new way of presenting potential reuse candidates to developers. Ye distinguishes two fundamentally contrary ways of getting information, the first one is information access, which is the classical “pull” approach where a user is actively browsing or searching for information, i.e. the system is reactive. The second approach is called information delivery and is based on “push” technology which monitors the activities of users and offers corresponding information which it considers useful in this context. Such systems are called *proactive*.

Although this proactive approach is very interesting, Ye's implementation is targeted to Emacs, a standard Unix text-editor, and not integrated into a modern IDE such as Eclipse. Furthermore, Ye only used a small repository with just a few hundred assets and performed no empirical evaluation. His results can only be regarded as anecdotal since he only had five subjects that tested his system and considered it useful. Furthermore, we believe it is very unlikely that the CodeBroker system would be able to scale up to repositories with millions of assets since its underlying retrieval technique is based on Latent Semantic Indexing [Dee90], which is well known to be effective in “understanding” natural language, but is also require high processing power. A further disadvantage is the need for developers to perform so-called “active commenting” (i.e. the developer has to explain what he wants to implement in extensive comments) of the class under development, which is required to provide the system with enough information to be able to find potential reuse candidates.

8.2.2 RASCAL

The most remarkable feature of RASCAL, a recommender agent for “agile reuse” developed by McCarey et al. at the University College in Dublin [McC07], is in fact that it tries to combine reuse with agile development, two ideas that at first sight inherently contradict each other. McCarey et al. suggest a way of promoting reuse in agile development through so-called “software recommendation” technology,

which is similar to CodeBroker. However, their “agile reuse” tool is an Eclipse plugin which uses collaborative and content-based filtering techniques [Bae99] to proactively suggest method invocations to developers. It does this by attempting to cluster Java objects according to the methods they use, just as Amazon, for example, clusters its customers according to the books they buy. The tool monitors method invocations in the class currently under development to predict method calls that are likely to be needed soon and suggests them to the developer. To evaluate their system the authors experimentally predicted invocations of the Java Swing Library in common open source systems and claim precision rates of around 30% for this setting. Although RASCAL showed good performance for the limited domain of Swing invocations, it is not clear whether this technique would work for other domains with many more classes that have much lower usage frequencies and how the system will scale up in general.

8.2.3 CODEGENIE

Up until very recently we were not aware of any approach coming close to the sophistication of our Extreme Harvesting technology. Although this is probably still the case, a group from the University of California working on the Sourcerer search engine (which is also contained in table 8.1 and briefly discussed in section 8.3) has presented an approach very similar to our Extreme Harvesting idea. They have called their tool CodeGenie and refer to their approach as Test-Driven Code Search (TDCS). Unfortunately, only an extended abstract [Lem07] and some videos on it are available so far. As far as we can tell from that, they are following a similar approach to us as demonstrated by the following figure:

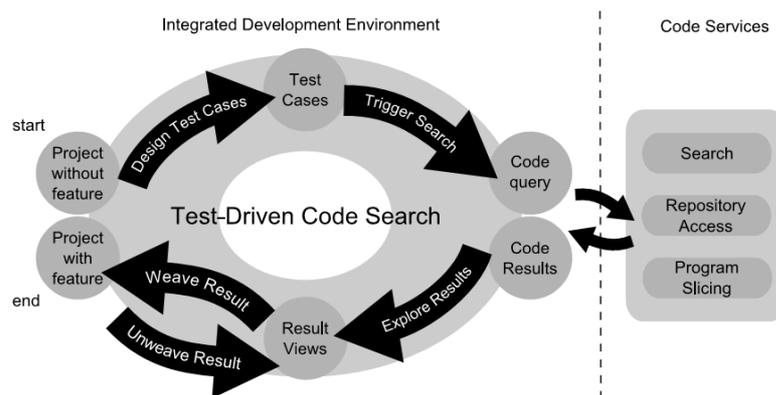


Figure 8.2: Process of a test-driven code search in CodeGenie.

CodeGenie is also an Eclipse plugin that generates a code query for Sourcerer out of a given test case. As far as we can judge, the developer then is able to inspect the delivered candidates (that are not yet tested automatically at that point) and to “weave” them into to the project under development. Once this has worked out, the project can be tested in the normal Eclipse environment to see whether the candidate fits into the project. It is also possible to “unweave” results again in order to integrate another candidate into the project. This approach obviously still requires a lot of human intervention and has not yet reached the degree of automation of Extreme Harvesting, but nevertheless it demonstrates the utility of test-driven reuse from another perspective.

8.2.4 AND MORE...

Since reuse research has a tradition of almost forty years and almost each approach has been accompanied by a prototypical implementation, there are dozens of other descriptions available in the

literature. However, none of these tools has made a big impact in the reuse community or is still available for testing today, let alone being in practical use somewhere. Most of these prototypes merely used simple information-retrieval methods and relied on extracting textual information from the components. Michail and Notkin [Mic99] have been worked with identifier (i.e. class and function) names to find similar components. Di Felice and Fonzi [Fel98] and Maarek et al. [Maa91] both developed a system which used the documentation of a component to automatically construct a reuse repository. Scott Henninger developed CodeFinder for his Ph.D. thesis [Hen93] at the University of Colorado in the early 1990s and used a hybrid technique. He emphasized query formulation and thus supported queries with reformulation as also described by [Fis89] for the general retrieval system Helgon. Inspired by [Moz84], Henninger used a retrieval algorithm based on the idea of spreading activation in AI applications. The publications by Rittri [Rit89] and Zaremski and Wing, already discussed in section 3.2.6, also developed prototypical systems that used signature matching for component retrieval. Another text-based retrieval prototype called ROSA was developed by Giradi and Ibrahim [Gir94]. Gomes et al. extended their so-called ReBuilder system with case-based reasoning based on a WordNet dictionary to reuse design patterns [Gom03]. The group of Silvio Meira in Brazil recently also put significant effort into the development of a reuse repository and accompanying plugin. However, their current Maracatu system supports only faceted (i.e. components can be searched by platform, component type and component model) and keyword-based searches. We regard the significance of the experiments described in [Gar06] as rather low since their repository was small (< 5,000 classes) and their relevance criterion was simply – as far as we found out in personal communication – the appearance of the search term or similar words in the delivered candidates.

Besides CodeBroker and RASCAL, which he have already explained in a separate subsection above, some other proactive recommendation tools have been recently presented in the literature. We have already briefly mentioned the work of Mandelin et al. [Man05] who created Prospector, an Eclipse plugin which is able generate a chain of required method calls in order to come to a desired return type from a given set of input parameter types. This can be particularly helpful for understanding complicated APIs such as the one of Eclipse more quickly. Holmes et al. [Hol06] followed a similar approach when they developed their Strathcona tool, which is also implemented as an Eclipse plugin that recommends helpful method execution chains derived for frameworks in its index. According to the evaluations presented in the publications, both tools perform well in this task as long as proprietary frameworks are used and it is certainly an interesting question whether this approach could be scaled to an open component collection. In addition to many tools surveyed in this section, a countless number of other less influential tools exists, but can not be mentioned any more for brevity. However, some other reuse tools deserve our attention because of their innovative ranking approaches and thus we explain them in more detail in the next section.

8.3 RESULT RANKING

All search engines, whether for code or for normal web pages, invariably try to rank the results that match a given query in order to offer the best results first. When potentially hundreds or even thousands of functionally matching components can be retrieved for a query it is of particular importance to order

these candidates to provide the ones with the best quality first. Mili et al. [Mil98] categorized ranking approaches as a topological retrieval method in their survey (explained in section 3.2.9). Unfortunately, relatively little work has been performed in this area so far, perhaps because there is no commonly accepted and easily obtainable notation for measuring the “distance” between components. Another rationale for this is not surprising at all: for the relatively small repositories utilized in the past the ranking of results was simply not necessary.

Some obvious approaches have been proposed in the literature in the past, but none of them reached a degree of maturity that would have made it usable in practice. Our literature survey and analyses have revealed approaches in the following three categories. As we briefly discussed in section 3.2.10 component ranking could be based on:

1. quality of service
2. popularity
3. distance to the query

Substantial work has only been performed for the second category in the context of component retrieval. The so-called ComponentRank approach of Inoue et al. [Ino05], based on the indirect measure of usage counting, has experienced significant attention in this area so far. The authors developed the so-called SPARS-J system which could be seen as the first serious attempt to construct a scalable reuse repository. SPARS-J uses a simple text-based retrieval approach and comprises about 180,000 components, i.e. Java classes in this case. Compared to earlier works in this area, this represents major progress in terms of repository size and has made the need for a ranking approach apparent. The authors drew some inspiration from PageRank [Pag98], which is the Google's way of ranking web pages and developed the ComponentRank algorithm by analogy. According to the Mili survey [Mil98] on retrieval techniques, the underlying information retrieval approach has a rather average level of precision and the authors argue that a ranking algorithm could remedy this issue. The idea is to rank the components according to their popularity, i.e. components which are used often by other components should appear highest in the list of results. Furthermore, components that have a higher ComponentRank (CR) than others have a higher impact on the calculation of the CR of the components they use than the ones with a lower CR. This relationship is stored in a directed (and weighted) graph as shown in figure 8.3 below.

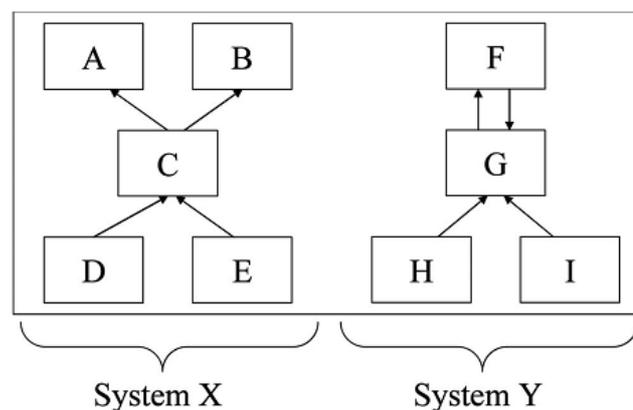


Figure 8.3: Representation of a system as a directed graph for ComponentRank [Ino05].

A node represents a component while an edge stands for a use relation. Relations considered by SPARS-J are:

- ◆ class inheritance
- ◆ interface implementation
- ◆ abstract class implementation
- ◆ variable declaration
- ◆ instance creation
- ◆ field access
- ◆ method invocation

A node v in the graph is assigned a non-negative weight $w(v)$ with $0 \leq w(v) \leq 1$. The rank of a component is derived from its weight. The higher the weight, the higher the rank. The weight of a node v_i is calculated by summing up the weight of all incoming edges:

$$w(v_i) = \sum_{e_{ki} \in \text{IN}(v_i)} w'(e_{ki}).$$

An edge's weight is given by the following formula:

$$w'(e_{ij}) = d_{ij} \times w(v_i)$$

d_{ij} is the so-called distribution ratio, i.e. a simple factor to distribute the weight of a node to its outgoing edges. Initially, all node weights are set to an arbitrary value and the weights are iteratively computed until they converge. Further details about the computation can be found in [Ino05].

SPARS-J is one of the few academic reuse repositories which has an accessible prototype on the web (<http://demo.spars.info>). The authors have performed evaluations of their system in which it performs well in comparison with general search engines such as Google. However, their experimental design contains some serious flaws since for example the relevance criterion is not made explicit and thus a replication of this experiment is made impossible. Furthermore, ComponentRank faces another inherent problem which is neither sufficiently investigated nor solved in their publications. While the original PageRank is based on URLs (i.e. Uniform Resource Locators), which are unique, Java classes are not necessarily unique. ComponentRank therefore requires a reliable mechanism to identify components since a repository can contain copies or various versions of the same component. While this might be possible with a few carefully selected version control repositories and source files, it becomes impossible as soon as binary files or components from the open web or automatically crawled repositories are included. Thus, ComponentRank is not likely to be applicable for comprehensive and diverse content of the kind stored in Merobase. The absence of this distinction between closeness of match and ranking is obviously another weakness of the ComponentRank algorithm, which tries to compensate for a rather weak name-based matching by incorporating the popularity of a result. However, this approach obviously does not overcome the problems of weak matching since even a very popular result might simply not be the right one for a query.

The Sourcerer system [Baj06] is another academic component search engine recently developed at the University of California in Irvine. It also has a special focus on the ranking and it tries to combine three values for it, namely the TFIDF (term frequency inverse document frequency [Bae99], i.e. the importance of the query term in a document) values delivered from Lucene, a so-called CodeRank value similar to ComponentRank and some other special heuristics. Unfortunately, at the time of writing, no detailed information was available.

9 EPILOGUE

The best way to predict the future is to invent it.

-- Alan Kay

9.1 SUMMARY

The central goal of this thesis was to deliver a solution for semantic software component retrieval that supports the vision of KobrA [Atk02] to select a component from a repository according to a given specification. At the beginning of our work we performed a thorough investigation of the state of the art in component- and service-based software reuse and whether and how it is integrated into mainstream development processes. We found a vast number of approaches that focussed on the idea of reusing pre-produced components in other development projects. However, none of them delivered a useful solution in practice, and none of them demonstrated how a potentially functioning solution would be integrated into common development processes. Our extensive literature survey revealed the following four major challenges for a practically usable component reuse solution:

1. The repository problem
2. The representation problem
3. The retrieval problem
4. The usability problem

Most approaches so far only deal with one or two of these problems and consequently have never reached a degree of maturity that would have satisfied practical demands.

Within the scope of this thesis, however, we developed a reuse system that tackles all of the above problems. It is based on a repository of almost 10 million components, which is more than a thousand times larger than any repositories available at the time when this thesis was started in 2004. It offers sophisticated component descriptions based on fields that allow a high-quality (pre-)selection of reusable candidates in typically under 5 seconds. The final assessment of fitness for purpose is performed with the help of our new operational retrieval approach based on standard unit tests. We have integrated a client for this repository into the well-known Eclipse development environment where it runs transparently in

the background and is able to generate queries from class stubs or even directly from unit tests. In the case of test-driven development, our system is able to propose fully tested and thus reusable candidates about minute after a test case has been specified by the developer.

Additionally, we have found, investigated and optimized a number of further use cases to address this issue. Finally, an evaluation of our ideas and a survey of related works published in the last four years round off the description of the work performed in the context of this thesis. The following subsection lists the contributions of this dissertation in more detail.

9.1.1 CONTRIBUTIONS

Based on the open issues in the area of component-based reuse which we identified in the first chapter, this subsection presents the general contributions and steps-forward that we were able to make within the scope of this dissertation to improve the state of the art in component-based software reuse. Referring to the quote of Carma McClure [McC97] that we cited in the first chapter, we briefly summarize our contributions in an abstract way before we give a more detailed discussion in the second part of this subsection. McClure identified the following three areas where progress is necessary to come to a practically usable reuse solution:

1. Something to reuse

We have shown that it is feasible to use the Internet or version control repositories of large companies as a source for software components. On top of this, early trials proved that it is indeed possible to use general web search engines to retrieve components with surprisingly good precision. From the sources collected during these investigations we were able to build one of the largest software repositories currently available containing more than 10 million entries and to carry out efficient searches on it with the help of the open source text search engine Lucene.

2. Software tool set

Recent research has proposed that effective reuse tools should be seamlessly integrated into development environments and recommend potentially reusable components pro-actively, i.e. without requiring the developer to issue searches manually. Consequently, we have built a pro-active plugin for the well-known Eclipse IDE that automatically extracting queries from class stubs or even test cases under development and presents reusable candidates unprompted. Furthermore, we have developed another recommendation algorithm which is able to derive commonly used operations related to a search term and thus could be used to support developers as early as during software design.

3. Software process

Most software development processes in the past did not care about reuse at all or if they did, they only delivered very vague guidelines on how to reuse. Our thesis proposes a test-driven reuse approach to reuse called Extreme Harvesting which is easily applicable in most agile development methodologies and even in test-first variants of the RUP, KobrA or most other modern development processes.

During our work we have identified another gap in the state of the art, namely, that the performance of component retrieval techniques has only been weakly investigated. Thus, we were also able to make a contribution in the area of evaluating reuse repositories:

4. Retrieval Evaluation

We have carried out a precision analysis of various component retrieval techniques based on a component specification comprising interface and test cases. This involved 4 million Java source components and revealed that some well established older techniques are no longer usable on such large repositories. On the other hand, we were able to show that interface-based retrieval could be used as a time-saving replacement for a specification-based approach such as Extreme Harvesting.

Furthermore, we have transferred older ideas such as signature matching to Java and other modern object-oriented languages for the first time and have defined an innovative data representation format that allows fast interface-based component retrieval and signature matching with common text retrieval systems such as Lucerne. To fully implement our Extreme Harvesting vision we have implemented a set of innovative algorithms such as a parser that extracts the interface of the class under test from JUnit test cases and a fully automated testing system that is purely signature based. In other words, it completely ignores class and method names and furthermore is able to permute through all possible “wirings” during the process of adapter creation until it finds a working solution. Since the classic Gang of Four adapter pattern is not sufficient for adapters that require the adapted class itself as a parameter, we also developed our so-called managed adapter that solves this problem and makes the automated creation of adapters for reusable components feasible.

Thus, in summary, we have developed a specification-based reuse system based on one of the largest reuse repositories so far in existence that is fully integrated into a modern development environment. We have shown that it has the potential to support reuse and thus to accelerate software development. The optimal hosting process for our approach is a test-driven development process (like most agile processes) since, according to our results, unit tests are a perfectly good starting point for component retrieval. Admittedly, one small limitation remains, we are not yet able to fully implement the vision of the Kobra development method which proposes hierarchically (de-)composable components. However, this is due to the current generation of programming languages since none of them contains sufficient for components.

9.2 FUTURE WORK

Extreme Harvesting is a highly successful proof of concept and opens up a whole lot of further research perspectives. We have shown that the idea is applicable for Java and web services, and see no reason why it should not be transferable to most other common programming languages. The most challenging question, however, is whether and how larger components or component ensembles can be retrieved. As we pointed out in section 5.6, current programming languages only support classes and packages, but not the concept of a component as envisaged e.g. by Kobra [Atk02]. Thus, currently, the discovery of “super-components” as a collection of smaller components typically requires a detailed specification of each of the smaller components until a description at the level of atomic units of the used programming

language is attained. This, of course, defeats most of the benefits of component-based development, but it is questionable whether it will become possible to reach a higher abstraction level with the current generation of programming languages. However, it is certainly also interesting to find whether there are any heuristics that would make it feasible to recognize cohesive component ensembles automatically.

Another part of our Extreme Harvesting approach that has the potential to be improved is the usage of linguistic elements (such as class and method names) for the initial result population. Although the search algorithms proposed in this thesis are very precise they are likely to be optimizable with e.g. synonyms or hypernyms optimized on programming in the case that few or no results are found for the initial request. However, this problem has been plaguing general information retrieval systems for years and to our experience, naively adding synonyms as search terms quickly leads to an explosion in the number of results and normally makes them unusable. We believe that a special thesaurus for names of software entities and a special decomposition of composite names (e.g. `IsACourseToBeScheduled`) might be more promising. However, first evaluations in this direction performed in the context of this thesis are quite disappointing [Gru07] and rather indicate that a splitting of composite class and method names might only become useful when combined with interface-driven searches or Extreme Harvesting. Thus, one goal for the near future should be the discovery of the optimal mix of heuristics that delivers an acceptable amount of tested results within a reasonable period of time.

In [Hum05c] we have already discussed the idea of using Extreme Harvesting as a source for back-to-back testing [Vou90] and n-version programming approaches [Avi95] where different versions no longer have to be laboriously developed but can be harvested from the web. The application of our repository to improve effort estimation approaches such as COCOMO [Boe00] also seems to be a feasible and interesting option.

From a practical point of view, there are other obvious ways of improving the Merobase system. We are already working on a more responsive version of our back-end system, which is able to index changes in a software repository in (near) real-time. A fast and syntax-aware search system as developed for this thesis is also likely to become a valuable extension for web service composition environments as currently promoted by SAP (Netweaver Composition Environment) and other companies. And certainly, to be able to assess the efficiency of our approach under practical conditions, the controlled application of our technology in an industrial setting is also desirable.

9.3 CONCLUDING VISION

Since testing still is (and will certainly remain for some time to come) the only means by which a software component can be judged as “fit for purpose”, we believe that, together with our test-driven reuse approach, it can become the central driver for component and service markets in the mid-term future. Thus, our basic idea is to integrate the ability of testing components into standard software search brokers, as we have done it with Merobase. As well as delivering components that syntactically match users’ queries, search engines enhanced in this way will also be able to execute tests on the user's behalf. In contrast with current testing approaches, however, a new form of “blind testing” is required to protect the interests of component providers and users in a commercial brokerage scenario. Thus, we propose a

form of testing in which the user is only provided with an indication of whether a test was passed or failed by a trusted broker, but not with the results generated by the component in the event of a failure [Hum06b]. Furthermore, it is important that the expected result of a test submitted by the user is also not disclosed to the component under test since it could otherwise be used to return spoofed results. The search engines in our vision thus acts as the trusted broker or mediator between component providers and users.

Since the overall effect is it to allow potential users to test components with minimal knowledge about them as black boxes, we refer to the overall model as black box brokerage (BBB). From the point of view of a component provider, a black-box broker is little different from a standard component repository such as a UDDI repository. The difference is that the component provider must provide all the information and content required to actually execute the component. In the case of a standard, embeddable component, such as a source code module (e.g. class) or a non-source component (e.g. Java Byte Code or .NET module) this means that the executable (or compilable) description must be provided. In the case of an online service, such as a web service, this means that a suitable account must be created and all necessary access keys provided. From the point of view of potential component users, the only difference between a black box broker and a normal component search engine is that once a component of interest has been identified (usually via a normal syntactic search), the user can supply one or more test cases, which the broker will apply to the component on the user's behalf. In the ideal case, a search could even be test-driven as we have developed it in this thesis.

Although the idea is simple, there are some significant challenges to be overcome in its implementation as we discussed in more detail in [Hum06b]. In a nutshell, these problems are the creation of a secure and efficient testing environment and an adapter that is capable of mapping a user's test cases to components with potentially different interfaces. However, these problems have already largely been solved within our test-driven reuse approach and thus it seems feasible to integrate both these approaches with relatively little effort. It appears feasible to offer a systems that recommends open source and even commercial components that are guaranteed to work without violating the interests of the respective component owners in the near future. The research conducted for this dissertation has paved the way to create component markets that are able to offer tested components appropriate for a given task without any additional effort for developers and without the risk to component producers of having the functionality exploited without payment.

10 REFERENCES

Half of knowledge is to know where to find it.
-- Michel de Montaigne

- [Abt98] Abts, C., B. Clark, S. Devnani-Chulani, E. Horowitz, R. Madachy, D. Reifer, R. Selby, and B. Steece, "COCOMO II model definition manual", Tech. Rep., Center for Software Engineering, USC, 1998, <http://sunset.usc.edu/COCOMOII/cocomox.html#downloads>.
- [Ami04] Amin, R., M. Ó Cinnéide and T. Veale: "LASER: A Lexical Approach to Analogy in Software Reuse", Proceedings of the International Workshop on Mining Software Repositories, Edinburgh, 2004.
- [Amb03] Ambler, S.W.: Agile Database Techniques - Effective Strategies for the Agile Software Developer, Wiley, 2003.
- [Ant04] Antoniou, G. and F. van Hamelen: "Web Ontology Language: OWL", in S. Staab and R. Studer (eds.): Handbook on Ontologies, Springer, 2004.
- [Atk95] Atkinson, S. and R. Duke: "A Methodology for Behavioural Retrieval from Class Libraries", Australian Computer Science Communications, Vol. 17, Iss. 1, 1995.
- [Atk02] Atkinson, C., J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J. Wüst, J. Zettel: Component-based Product Line Engineering with UML, Addison Wesley, 2002.
- [Atk04] Atkinson, C. and O. Hummel: "Towards a Methodology for Component-Driven Design", Proceedings of the International Workshop on Rapid Integration of Software Engineering Techniques (appeared in LNCS 3475), 2004.
- [Atk07] Atkinson, C., D. Brenner, P. Bostan, G. Falcone, M. Gutheil, O. Hummel, M. Juhasz and D. Stoll: "Modeling Components and Component-Based Systems in KobrA", in A.
-

- Rausch, R. Reussner, R. Mirandola, F. Plasil (eds.): *The Common Component Modeling Example: Comparing Software Component Models*, Springer, 2007.
- [Avi95] Avizienis, A: "The Methodology of N-Version Programming" in *Software Fault Tolerance*, John Wiley & Sons, 1995.
- [Bae99] Baeza-Yates R. and B. Ribeiro-Neto: *Modern Information Retrieval*, Addison-Wesley, 1999.
- [Baj06] Bajracharya, S., T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. Lopes: "Sourcerer: a search engine for open source code supporting structure-based search", *Proceeding of the Conference on Object Oriented Programming Systems Languages and Applications*, 2006.
- [Bar06] Baroudi, C. and F. Halper: "SOA Implementation Satisfaction", Technical Report, Hurwitz and Associates, 2006.
- [Bas86] Basili, V.R., R.W. Selby, D.H. Hutchens: "Experimentation in Software Engineering", *IEEE Transactions on Software Engineering*, Vol. 12, Iss. 7, 1986.
- [Bas88] Basili, V.R., D. Rombach: "Towards a Comprehensive Framework for Reuse: A reuse-enabling software evolution environment", *Proceedings of the NASA Goddard Flight Center Software Engineering Workshop*, 1988.
- [Bas91] Basili, V.R., D. Rombach: "Support for Comprehensive Reuse", Technical Report CS-TR-2606, University of Maryland, 1991.
- [Bas96] Basili, V.R., S. Green, O. Laitenberger, F. Lanubile, F. Shull, S. Sorumgard, M.V. Zelkowitz: "The Empirical Investigation of Perspective-Based Reading", *Journal of Empirical Software Engineering*, Vol. 1, Iss 2, 1996.
- [Bas03] Bast, W., A.G. Kleppe and J.B. Warmer: *MDA Explained: The Model Driven Architecture: Practice and Promise*, Addison-Wesley, 2003.
- [Bau05] Baumann, S. and O. Hummel: "Enhancing Music Recommendation Algorithms Using Cultural Metadata", *Journal of New Music Research*, Vol. 34, Iss. 2, 2005.
- [Bec99] Beck, K.: *Extreme Programming Explained: Embrace Change*, Addison-Wesley, 1999.
- [Bec99b] Beck, K. and E. Gamma: "JUnit: A Cook's Tour", *JavaReport*, Iss. 8, 1999.
- [Bec03] Beck, K.: *Test-Driven Development by Example*, Addison-Wesley, 2003.
- [Bei90] Beizer, B: *Software Testing Techniques*, International Thomson Press, 1990.
- [Bei95] Beizer, B.: *Black-Box-Testing: Techniques for Functional Testing of Software and Systems*, Wiley, 1995.
- [Ber99] Berners-Lee, T. : *Weaving the Web*, Texere Publishing, 1999.
-

-
- [Ber01] Berners-Lee, T., J. Hendler, O. Lassila: "The Semantic Web", Scientific American, Vol. 284, Iss. 5, 2001.
- [Bla82] Blalock, A.B. and H.M. Blalock: Introduction to Social Research, second edition, Prentice Hall, 1982.
- [Boe76] Boehm, B.W.: "Software Engineering", IEEE Transactions on Computers, 1976.
- [Boe84] Boehm, B.: "Verifying and validating software requirements and design specifications", IEEE Software, Vol. 1 (1), 1984.
- [Boe88] Boehm, B.W.: "A Spiral Model of Software Development and Enhancement", IEEE Computer Vol 21, Iss. 5, 1988.
- [Boe00] Boehm, B.W., B. Steece and R. Madachy: Software Cost Estimation with Cocomo II, Prentice Hall, 2000.
- [Boo87] Booch, G.: Software Components with Ada: Structures, Tools and Subsystems, Benjamin-Cummings, 1987.
- [Box97] Box, D.: Essential COM: The Component Object Model, Addison-Wesley, 1997.
- [Bri01] Briand, L. and Y. Labiche: "An UML-Based Approach to System Testing", Proceedings of the International Conference on Modeling Languages, Concepts and Tools, 2001.
- [Bri03] Brickley, D. and R. Guha: RDF Vocabulary Description Language 1.0: RDF Schema, W3C, 2003.
- [Bro87] Brooks, F. P., "No Silver Bullet - Essence and Accident in Software Engineering", Computer 20, April 1987.
- [Bro98] Brown, W.J., R.C. Malveau and H. McCormick: Anti-Patterns. Refactoring Software, Architecture and Projects in Crisis, Wiley, 1998.
- [Bro02] Brown, A.W. and G. Booch: "Reusing Open-Source Software and Practices: The Impact of Open-Source Software on Commercial Vendors", C. Gacek (Ed.): LNCS 2319, Springer, 2002.
- [Bro05] Broy, M. and A. Rausch: "Das neue V-Modell XT" (in german), Informatik Spektrum, Vol. 28, Iss. 3, 2005.
- [Bus96] Buschmann, F., R. Meunier, H. Rohnert, H., P. Sommerlad and M. Stal: Pattern-Oriented Software Architecture, Wiley, 1996.
- [Cal91] Caldiera, G. and V.R. Basili: "Identifying and Qualifying Reusable Software Components", IEEE Computer, Vol. 24, Iss. 2, 1991.
- [Car05] Cardoso, J. and A. Shet: "Introduction to Semantic Web Services and Web Process Composition", LNCS 3387, Springer, 2005.
-

- [Car01] Carey, J. and B. Carlson: "Business components", Component-based Software Engineering: Putting the Pieces Together, G.T. Heinemann and W.T. Council (eds.), Addison-Wesley, 2001.
- [Cha94] Chan, L.M.: Dewey Decimal Classification, Forest Press, 1994.
- [Che00] Cheesman, J and J. Daniels: UML Components: A Simple Process for Specifying Component-Based Software, Addison-Wesley, 2000.
- [Cho96] Chou, S.C., J.Y. Chen and C.G. Chung: "A Behavior-Based Classification and Retrieval Technique for Object-Oriented Specification Reuse", Journal for Software Practice and Experience, Vol. 26, Iss. 7, 1996.
- [Chr03] Chrissis, M.B., M. Konrad and S. Shrum: CMMI. Guidelines for Process Integration and Product Improvement, Addison-Wesley, 2003.
- [Cle95] Clements, P.: "From Subroutines to Subsystems: Component-Based Software Development", The American Programmer, Vol. 8, Iss. 11, 1995.
- [Cle02] Clements, P. and L. Northrop: Software Product Lines: Practices and Patterns, Addison-Wesley, 2002.
- [Cli07] Clinton, D.: OpenSearch Specifications, 1.1, Draft 3, A9.com, 2007.
- [Coa90] Coad, P. and E. Yourdon: Object-Oriented Analysis, second edition, Prentice Hall, 1990.
- [Coa91] Coad, P. and E. Yourdon: Object-Oriented Design, Prentice Hall, 1991.
- [Coc01] Cockburn, A.: Agile Software Development, Addison Wesley, 2001.
- [Col94] Coleman, D., P. Arnold, S. Bodoff, C Dollin, H. Gilchrist, F. Hayes and P. Jeremaes: "Object-Oriented Development: The Fusion Method", Prentice-Hall, 1994.
- [Cor01] Cormen, T., C. Leiserson, R. Rivest, C. Stein: Introduction to Algorithms, 2nd Edition, MIT Press, 2001.
- [Crn06] Crnkovic, I., M. Chaudron and S. Larsson: "Component-based Development Process and Component Lifecycle", Proceedings of the International Conference on Software Engineering Advances, 2006.
- [Cza00] Czarnecki, K., U.W. Eisenecker: Generative Programming, Springer, 2000.
- [Dah66] Dahl, O.J. and K. Nygaard: "SIMULA: an ALGOL-based simulation language", Communications of the ACM, Vol. 9, Iss. 9, 1966.
- [Dav95] Davey, N., P. Barson, S.D.H. Field, R. J. Frank, D.S.W. Tansley: "The Development of a Software Clone Detector". International Journal of Applied Software Technology, Volume 1 Number 3/4, 1995.
-

-
- [Dee90] Deerwester, S., S.T. Dumais, G. W. Furnas, T.K. Landauer and R. Harshman: "Indexing by Latent Semantic Analysis", *Journal of the American Society for Information Science*, Vol. 41, Iss. 6, 1990.
- [Des06] Desouza, K.C., Y. Awazu, A. Tiwana: "Four Dynamics for bringing use back into software reuse", *Communications of the ACM*, Vol. 49, Iss. 1, 2006.
- [Dij70] Dijkstra, E.W.: *Notes on Structured Programming*, Technological University Eindhoven, 1970.
- [Dij72] Dijkstra, E.W.: "The Humble Programmer", *Communications of the ACM*, Vol. 15, Iss. 10, 1972.
- [Dog05] Dogpile.com: "Different Engines, Different Results", Technical Report: <http://com-paresearchengines.dogpile.com/OverlapAnalysis.pdf> (accessed 09/08/2005).
- [Eck06] Eckel, B.: *Thinking in Java*, Prentice Hall, 2006.
- [Enc02] Marciniak, J.J.: *Encyclopedia of Software Engineering*, Second Edition, Wiley, 2002.
- [End03] Endres, A. and D. Rombach: *A Handbook of Software and Systems Engineering, Empirical Observations, Laws, and Theories*, Person Education, 2003.
- [Faf94] Fafchamps, D.: "Organizational Factors and Reuse", *IEEE Software* Vol. 11, Iss. 5, 1994.
- [Fel98] Di Felice, P. and G. Fonzi: "How to write Comments suitable for Automatic Software Indexing", *Journal for Systems and Software*, Vol. 42, Iss. 1, 1998.
- [Fen05] Fensel, D., J. Hendler, H. Lieberman, W. Wahlster (eds.): *Spinning the Semantic Web: Bringing the World Wide Web to its Full Potential*, MIT Press, 2005.
- [Fis89] Fischer, G. and H. Nieper-Lemke: "Helgon: Extending the Retrieval by Reformulation Paradigm", *Proceedings of the International Conference on Human Factors in Computing Systems*, 1989.
- [Fis91] Fischer, G., S. Henninger and D. Redmiles: "Cognitive Tools for Locating and Comprehending Software Objects for Reuse", *Proceedings of the International Conference on Software Reuse*, 1991.
- [Fis98] Fischer, G.: "Seeding, Evolutionary Growth and Reseeding: Constructing, Capturing and Evolving Knowledge in Domain-Oriented Design Environments", *Journal of Automated Software Engineering*, Vol. 5, Iss. 4, 1998.
- [Fow99] Fowler, M.: *Refactoring*, Addison-Wesley, 1999.
- [Fow01] Fowler, M. and J. Highsmith: "The Agile Manifesto", *Journal for Software Development*, Vol. 9, Iss. 8, 2001.
-

- [Fow03] Fowler, M.: UML Distilled: A Brief Guide to the Standard Object Modeling Language, Addison-Wesley, 2003.
- [Fra87] Frakes, W.B. and B.A. Nejmeh: "An Information System for Software Reuse", in W. Tracz (ed.): Software Reuse: Emerging Technology, Computer Society Press, 1987.
- [Fra94] Frakes, W.B. and T.P. Pole: "An Empirical Study of Representation Methods for Reusable Software Components", IEEE Transactions on Software Engineering Vol. 20, Iss. 8, 1994.
- [Fra95] Frakes, W.B. and C.J. Fox: "Sixteen Questions about Software Reuse", Communications of the ACM, Vol 38 Issue 6, 1995.
- [Fra96] Frakes, W.B. and C. Terry: "Software Reuse: Metrics and Models", ACM Computing Surveys, Vol. 28, No. 2, 1996.
- [Fra96b] Frakes, W.B. and C.J. Fox: "Quality Improvement Using a Software Reuse Failure Modes Model", IEEE Transactions on Software Engineering Vol. 22, Iss. 4, 1996.
- [Fra05] Frakes, W.B. and K. Kang: "Software Reuse Research: Status and Future", IEEE Transactions on Software Eng., Vol. 31, No. 7, 2005.
- [Fur87] Furnas, G.W., T.K. Landauer, L.M. Gomez and S.T. Dumais: "The Vocabulary in Human-System Communication", Communications of the ACM, Vol. 30, Iss. 11, 1987.
- [Gaf89] Gaffney, J.E. and T.A. Durek: "Software Reuse – Key to Enhanced Productivity: Some Quantitative Models", Information and Software Technology, Vol 31, Iss. 5, 1989.
- [Gar06] Garcia, V. C., D. Lucrédio, F.A. Durão, E.C.R. Santos, E.S. Almeida, R.P. Fortes, S.R.L. Meira: "From Specification to the Experimentation: A Software Component Search Engine Architecture", Proceedings of the International Symposium on Component-Based Software Engineering (CBSE), 2006.
- [Gil93] Gilb, T., D. Graham, S. Finzi: Software Inspections, Addison Wesley, 1993.
- [Gil97] Giloi, W.: "Konrad Zuse's Plankalkül: The First High-Level 'non von Neumann' Programming Language", IEEE Annals of the History of Computing, Vol. 19, Iss. 2, 1997.
- [Gir94] Girardi, M.R. and B. Ibrahim: "A Similarity Measure for Retrieving Software Artifacts", Proceedings of the International Conference on Software Engineering and Knowledge Engineering, 1994.
- [GoF95] Gamma, E., R. Helm, R. Johnson, and J. Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.
- [Gom03] Gomes, P., F.C. Pereira, P. Paiva, N. Seco, P. Carreiro, J. Ferreira and C. Bento: "Selection and Reuse of Software Design Patterns using CBR and WordNet", Proceedings of the International Conference on Software Engineering and Knowledge Engineering, 2003.
-

-
- [Gre03] Greenfield, J. and K. Short: "Software Factories: Assembling Applications with Patterns, Models, Frameworks and Tools", ACM Press, 2003.
- [Gri93] Griss, M.: "Software Reuse: From Library To Factory", IBM Systems Journal, Vol. 32, Iss. 4, 1993.
- [Gri94] Griss, M., J. Favaro and P. Walton, "Managerial and Organisational Issues: Starting and Running a Software Reuse Program", Software Reusability, W. Schaefer, R. Prieto-Diaz and M. Matsumoto (eds.), Horwood, New York, 1994.
- [Gru07] Grunert, M.: "Semantic Component Search Using Latent Semantic Indexing", Diploma Thesis, University of Mannheim, 2007.
- [Gsc02] Gschwind, T.: "Adoption and Composition Techniques for Component-Based Software Engineering", Ph.D. Thesis, Technical University of Vienna, 2002.
- [Gul05] Gulli, A. and A. Signorini: "The Indexable Web is more than 11.5 Billion Pages", Proceedings of the International World Wide Web Conference, 2005.
- [Hal93] Hall, R. J.: "Generalized Behavior-Based Retrieval", Proceedings of the International Conference on Software Engineering, Baltimore, United States, 1993.
- [Hal77] Halstead, M.H.: Elements of Software Science, Elsevier, 1977.
- [Ham02] Hamlet, R.: "Random Testing", Encyclopedia of Software Engineering, Wiley, Second Edition, 2002.
- [Hat04] Hatcher, E. and O. Gospodnetic: Lucene in Action, Manning, 2004.
- [Hen81] Henry, S. and D. Kafura: "Software Structure Metrics Based on Information Flow", IEEE Transactions on Software Engineering, Vol. 7, Iss. 5, 1981.
- [Hen93] Henninger, S.: "Locating Relevant Examples for Example-Based Software Design", Ph.D. Dissertation, Department of Computer Science, University of Colorado, USA, 1993.
- [Hol06] Holmes, R., R.J. Walker and G.C. Murphy: "Approximate structural context matching: An approach for recommending relevant examples", IEEE Transactions on Software Engineering, Vol. 32, Iss. 12, 2006.
- [Hum03] Hummel, O.: "Ermittlung von Musikähnlichkeit auf Basis von Community Features" (in German), Master's Thesis, Technical University of Kaiserslautern, 2003.
- [Hum04] Hummel, O. and C. Atkinson: "Extreme Harvesting: Test Driven Discovery and Reuse of Software Components", Proceedings of the International Conference on Information Reuse and Integration (IEEE-IRI), Las Vegas, USA, 2004.
- [Hum05a] Hummel, O. and C. Atkinson: "Automated Harvesting of Test Oracles for Reliability Testing", Proceedings of the First International Workshop on Testing and Quality
-

- Assurance for Component-Based Systems (TQACBS in Conjunction with COMPSAC), Edinburgh, Scotland, 2005.
- [Hum05c] Hummel, O., C. Atkinson, D. Brenner and S. Keklik: "Improving Testing Efficiency through Component Harvesting", Proceedings of the Brazilian Workshop on Component Based Development, 2006.
- [Hum06] Hummel, O. and C. Atkinson: "Using the Web as a Reuse Repository", Proceedings of the International Conference on Software Reuse, 2006.
- [Hum06b] Hummel, O., P. Bostan and C. Atkinson: "Towards the Automated Selling of Web Services over the Internet", Proceedings of the International Workshop for Technology, Economy, Social and Legal Aspects of Virtual Goods, 2006.
- [Hum06c] Hummel, O., C. Atkinson, D. Brenner and S. Keklik: "Improving Testing Efficiency through Component Harvesting", in Proceedings of the Brazilian Workshop on Component Based Development, 2006.
- [IEE83] IEEE: Glossary of Software Engineering Terminology, ANSI/IEEEStd. 729-1983.
- [Ino05] Inoue, K., R. Yokomori, H. Fujiwara, T. Yamamoto, M. Matsushita, S. Kusumoto.: "Ranking Significance of Software Components Based on Use Relations", IEEE Transactions on Software Eng., Vol. 31, No. 3, 2005.
- [Iso92] Isoda, S.: "Experience report on software reuse project: its structure, activities, and statistical results", Proceedings of the International Conference on Software Engineering, 1992.
- [Jan07] Janjic, W.: "Realizing High-Precision Component Recommendations for Software Development Environments", Diploma Thesis, University of Mannheim, 2007.
- [Jen95] Jeng, J.J. and B.H.C. Cheng: "Specification matching for software reuse: a foundation", ACM SIGSOFT Software Engineering Notes, Vol. 20, 1995.
- [Kal03] Kalfoglou, Y. and M. Schorlemmer: "Ontology Mapping: the state of the art", The Knowledge Engineering Review, Vol. 18, Iss. 1, 2003.
- [Kan76] Kantor, P.B.: "Availability Analysis", Journal of the American Society for Information Science, Vol. 27, 1976.
- [Kel99] Keller, R.K., R. Schauer, S. Robitaille and P. Pagé: "Pattern-based Reverse Engineering of Design Components", Proceedings of the International Conference on Software Engineering, 1999.
- [Kik05] Kiko, K.: "Towards a Unified Knowledge Representation Framework", Diploma Thesis, University of Mannheim, 2005.
-

-
- [Kic97] Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.M. Loingtier and J. Irwin: "Aspect-Oriented Programming", Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland, 1997.
- [Kim99] Kim, Y.G.; H.S. Hong, D.H. Bae and S.D. Cha: "Test cases generation from UML state diagrams", IEE Proceedings Software, Vol. 146, Iss. 4, 1999.
- [Kni86] Knight, J.C. and N.G. Leveson: "An Experimental Evaluation of the Assumption of Independence in Multi-Version Programming", IEEE Transaction on Software Engineering, Vol. 12, Iss. 1, 1986.
- [Kra03] Kratz, B.: "Empirical Research on the Relationship between Functionality and Interfaces of Software Components", Master's Thesis, Tilburg University, 2003.
- [Kru92] Krueger, C.W.: "Software Reuse", ACM Computing Surveys, Vol. 24, Iss. 2, 1992.
- [Kru00] Kruchten, P.: "The Rational Unified Process - An Introduction", 2nd edition, Addison Wesley, 2000.
- [Kru01] Kruchten, P.: "The Nature of Software: What's so Special about Software Engineering?", The Rational Edge, October 2001.
- [Kru07] Krug, M.: "FAST: An Eclipse Plug-In for Test-Driven Reuse", Diploma Thesis, University of Mannheim, 2007.
- [Lam98] Lam, W.: "A Case Study of Requirements Reuse through Product Families", Annals of Software Engineering, Vol. 5, 1998.
- [Lan87] Landauer, G. W., T. K. Furnas, L. M. Gomez, S. T. Dumais: "The Vocabulary Problem in Human-System Communication", Communications of the ACM, Vol. 30, Iss. 11, 1987.
- [Lap96] Laprie, J.C. and K. Kanoun: "Software Reliability and System Reliability", in M. Lyu (Ed.): Handbook of Software Reliability Engineering, McGraw-Hill, 1996.
- [Lar05] Larman, C.: Applying UML and Patterns - An Introduction to Object-Oriented Analysis and Design and Iterative Development, 3rd edition, Prentice Hall, 2005.
- [Las99] Lassila, O., R.R. Swick and others: Resource Description Framework (RDF) Model and Syntax Specification, W3C, 1999.
- [Lem07] Lems, O.A.L., S. Bacjracharya and J.Ossher: "CodeGenie: a Tool for Test-Driven Source Code Search", Extended Abstract in Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages and Applications, 2007.
- [Len87] Lenz, M., H. Schmid, P.W. Wolf: "Software Reuse through Building Blocks", in W. Tracz (ed.): Software Reuse: Emerging Technology, Computer Society Press, 1987.
-

- [Lew07] Lewandowski, D.; Höchstötter, N.: "Qualitätsmessung bei Suchmaschinen – System- und Nutzerbezogene Evaluationsmaße" (in German), *Informatik-Spektrum*, Vol. 30, Iss. 3, 2007.
- [Lie80] Lientz, B.P. and E.B. Swanson: *Software Maintenance Management*, Addison-Wesley, 1980.
- [Lis93] Liskov, B. and J.M. Wing: "Family Values: A Semantic Notion of Subtyping", Technical Report 562, Massachusetts Institute of Technology, 1993.
- [Llo04] Llorens, J., M. Fuentes and J. Morato: "UML Retrieval and Reuse Using XMI", *Proceedings of IASTED Conference on Software Engineering*, 2004.
- [Luc04] Lucrecio, D., A.F. Prado, E.S. de Almeida: "A Survey of Component Search and Retrieval", *Proceedings of the Euromicro Conference*, 2004.
- [Maa91] Maarek, Y.S., D.M. Berry and G.E. Kaiser: "An Information Retrieval Approach for automatically constructing Software Libraries", *IEEE Transactions on Software Engineering*, Vol. 17, Iss. 8, 1999.
- [Mad01] Madhavan, J., P. Bernstein and E. Rahm: "Generic Schema Matching with Cupid", *Proceedings of the 27th Conference on Very Large Scale Databases (VLDB)*, Roma (Italy), 2001.
- [Man05] Mandelin, D. L. Xu, R. Bodik and D. Kimelman: "Jungloid mining: helping to navigate the API jungle", *Proceedings of the Conference on Programming Language Design and Implementation*, 2005.
- [Man07] Manning, C.D. and P. Raghavan and H. Schütze: *Introduction to Information Retrieval* Cambridge University Press, 2007.
- [McC76] McCabe, T.J.: "A Complexity Measure", *IEEE Transactions on Software Engineering*, Vol. 2, Iss. 4, 1976.
- [McC07] McCarey, F., M. O' Cinneide and N. Kushmerick: "Knowledge Reuse for Software Reuse", *Journal for Web Intelligence and Agent Systems*, Vol. 1, Iss. 1, 2007.
- [McC97] McClure, C.: *Software Reuse Techniques: Adding Reuse to the System Development Process*, Prentice Hall, 1997.
- [McI68] McIlroy, D.: "Mass-Produced Software Components", *Software Engineering: Report of a conference sponsored by the NATO Science Committee*, Garmisch, Germany, 1968.
- [Men05] Mennecke, T.: "eDonkey2000 Nearly Double the Size of FastTrack", <http://www.slyck.com/news.php?story=814>, 2005.
- [Mey92] Meyer, B.: "Applying Design by Contract", *IEEE Computer*, Vol. 25, Iss. 10, 1992.
-

-
- [Mez01] Mezini, M., L. Seiter and K. Lieberherr: "Component Integration with Pluggable Composite Adapters", in M. Aksit: *Software Architectures and Component Technology*, Kluwer, 2001.
- [Mic99] Michail, A. and D. Notkin: "Assessing Software Libraries by Browsing similar Classes, Functions and Relationships", *Proceedings of the International Conference on Software Engineering*, 1999.
- [Mil90] Miller, G.A., R. Beckwith, C. Fellbaum, D. Gross, K. Miller: "Introduction to Wordnet: An On-Line Lexical Database", *CSL Report 43*, Princeton University, 1990, revised 1993.
- [Mil98] Mili, A., R. Mili and R. Mittermeir: "A Survey of Software Reuse Libraries", *Annals of Software Engineering* 5, 1998.
- [Mil99] Mili, A., S. Yacoub, E. Addy and H. Mili: "Toward an Engineering Discipline of Software Reuse", *IEEE Software*, Vol. 16, No. 5, 1999.
- [Mil02] Mili, H., A. Mili, S. Yacoub and E. Addy: *Reuse-Based Software Engineering – Techniques, Organisations and Controls*, Wiley, 2002.
- [Moh04] Mohagheghi, P: "The Impact of Software Reuse and Incremental Development on the Quality of Large Systems", Ph.D. Dissertation, Norwegian University of Science and Technology.
- [Moi92] Moineau, T. and M.C. Gaudel: "Software Reusability through Formal Specifications", in *Proceedings of a Workshop on Methods and Tools for Software Reuse*, 1992.
- [Mor02] Morisio, M., M. Ezran and C. Tully: "Success and Failure Factors in Software Reuse", *IEEE Transactions on Software Engineering*, Vol. 26, Iss. 4, 2002.
- [Moz84] Mozer, M.C.: "Inductive Information Retrieval using Parallel Distributed Computation", *Technical Report No. 8406*, Institute for Cognitive Science, San Diego, 1984.
- [Mye02] Myers, G.J.: *The Art of Software Testing*, 2nd edition, Wiley, 2002.
- [Nea96] Neal, L.: "Support for Software Design, Development and Reuse through an Example-Based Environment", in G. Szwillus & L. Neal (eds.), *Structure-Based Editors and Environments*, Academic Press, San Diego, 1996.
- [New02] Newcomer, E.: *Understanding Web Services, XML, WSDL, SOAP and UDDI*, Addison-Wesley, 2002.
- [Nun90] Nunamaker, J.F., M. Chen: "Systems Development in Information Systems Research", *Proceedings of the International Conference on Systems Sciences*, 1990.
- [OMG00] OMG: *The Common Object Request Broker: Architecture and Specification, Version 2.4*, Object Management Group, 2000.
-

- [OMG03] OMG: Unified Modelling Language Specification, Version 1.5, Object Management Group, 2003.
- [OMG04] OMG: Unified Modelling Language Specification, Version 2.0, Object Management Group, 2004.
- [OMG04b] OMG: Reusable Asset Specification, Object Management Group, 2004.
- [ORe05] O'Reilly, T.: "What is Web 2.0", O'Reilly Media, <http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html>, last accessed 11/07.
- [Ost92] Ostertag, E.J.: "A Classification System for Software Reuse", Ph.D. Dissertation, University of Maryland, 1992.
- [Owe86] Owen, D.: "Answers First, Then Questions", in D. A. Norman and S. W. Draper, (eds.), User Centered System Design, New Perspectives on Human-Computer Interaction, Erlbaum, 1986.
- [Pag98] Page, L., S. Brin, R. Motwani, T. Winograd: "The Pagerank Algorithm: Bringing Order to the Web", Proceedings of the International Conference on the World Wide Web, 1998.
- [Par72] Parnas, D.L.: "On the Criteria to be Used in Decomposing Systems into Modules", Communications of the ACM Vol. 15, No. 12, 1972.
- [Pau94] Paul, S. and A. Prakash: "A Framework for Source Code Search Using Program Patterns," IEEE Transactions on Software Engineering, Vol. 20, Iss. 6, 1994.
- [Pen99] Penix, J. and P. Alexander: "Efficient Specification-Based Component Retrieval", Journal for Automated Software Engineering, Vol. 6, Iss. 2, 1999.
- [Per93] Perry, D. and S. Popovich: "Inquire: Predicate-Based Use and Reuse", Proceedings of the Knowledge-Based Software Engineering Conference, 1993.
- [Pod93] Podgurski, A., L. Pierce: "Retrieving Reusable Software by Sampling Behavior", ACM Transactions on Software Engineering and Methodology, Vol. 2, Iss. 3, 1993.
- [Por06] Porter, M.F.: "An Algorithm for Suffix Stripping", Information Systems, Vol. 40, Iss. 3, 2006.
- [Pou95] Poulin, J. and K.J. Werkman: "Melding structured abstracts and World Wide Web for retrieval of reusable components", Proceedings of the Symposium on Software Reusability, 1995.
- [Pou99] Poulin, J.: "The Foundation of Reuse", position paper in proceedings of the 9th Annual Workshop on Software Reuse, Austin, USA, 1999.
-

-
- [Pou99b] Poulin, J.: "Reuse: Been There, Done That.", *Communications of the ACM*, Vol. 42, Iss. 5, 1999.
- [Pri87] Prieto-Díaz, R. and P. Freeman: "Classifying Software for Reusability", *IEEE Software*, Vol. 4, Iss. 1, 1987.
- [Pri91] Prieto-Díaz, R.: "Implementing faceted classification for software reuse" *Communications of the ACM*, Volume 34, Issue 5, 1991.
- [Pri91b] Prieto-Díaz, R. and G. Arango: *Domain Analysis and Software Systems Modeling*, IEEE Computer Society Press, 1991.
- [Ray97] Raymond, E.: "The Cathedral and the Bazaar", *First Monday* Vol. 3, Iss. 3, 1998: http://www.firstmonday.dk/issues/issue3_3/raymond, last accessed 12/05.
- [Ric78] Rich, C. and H.E. Schrobe: "An Initial Report on Lisp's Programmer's Apprentice," *IEEE Transactions on Software Engineering* Vol. 4, Iss. 11, 1978.
- [Rit89] Rittri, M.: "Using Types as Search Keys in Function Libraries", *Journal of Functional Programming*, Vol. 1, Iss. 1, 1989.
- [Roy70] Royce, W. W.: "Managing the Development of Large Software Systems", *Proceedings of the 9th. International Conference of Software Engineering*, 1970.
- [Sal75] Salton, G., A. Wong and C.S. Yang: "A vector space model for automatic indexing", *Communications of the ACM*, Vol. 18, 1975.
- [Sca02] Scacchi, W.: "Process Models in Software Engineering", J.J. Marciniak (Ed.): *Encyclopedia of Software Engineering*, Second Edition, Wiley, 2002.
- [Sam97] Sametinger, J.: *Software Engineering with Reusable Components*, Springer, 1997.
- [Sch99] Schmidt, D.: "Why Software Reuse has Failed and How to Make it Work for You", *C++ Report*, Vol 11, Iss. 1, 1999.
- [Sea98] Seacord, R.C., S.A. Hissam and K.C. Wallnau: "AGORA: a search engine for software components", *IEEE Internet Computing*, Vol. 2, Iss. 6, 1998.
- [Sea99] Seacord, R.: "Software Engineering Component Repositories", *Proceedings of the International Workshop on Component-Based Software Engineering*, 1999.
- [Shi07] Shi, X.: "Semantic Web Services: An Unfulfilled Promise", *IEEE IT Professional*, August 2007.
- [Som06] Sommerville, I.: *Software Engineering*, Addison-Wesley, 8th Edition, 2006.
- [Son07] Song, H., D. Cheng, D., A. Messer and S. Kalasapur: "Web Service Discovery Using General-Purpose Search Engines", *Proceedings of the International Conference on Web Services*, 2007.
-

- [Sou98] D'Souza, D.F. and A.C. Wills: *Object, Components and Frameworks with UML: The Catalysis Approach*, Addison-Wesley, 1998.
- [Ste01] Steele, R.: "Techniques for Specialized Search Engines", *Proceedings of the Conference on Internet Computing*, 2001.
- [Str94] Stringer-Calvert, D.W.J.: "Signature Matching for Ada Software Reuse", *Master's Thesis*, University of York, 1994.
- [Sun01] Sun: *EJB 2.0 Specification*, Sun, 2001.
- [Sun06] Sun: *EJB 3.0 Specification*, Sun, 2006.
- [SWE04] Abran, A., J.W. Moore, P. Bourque, R. Dupuis (Eds.): *Guide to the Software Engineering Body of Knowledge: SWEBOK*, IEEE Computer Society, 2004.
- [Szy02] Szyperski, C.: *Component Software*, Addison-Wesley, 2nd Edition, 2002.
- [Tur02] Turowski, K. (ed.): *Vereinheitlichte Spezifikation von Fachkomponenten: Memorandum des Arbeitskreises komponentenorientierte betriebliche Anwendungssysteme* (in german), Universität Augsburg, 2002.
- [Van07] Vanderlei, T.A., F.A. Duraõ, A.C. Martins, V.C. Garcia, E.S. Almeida and S. Meira: "A Cooperative Classification Mechanism for Search and Retrieval of Software Components", *Proceedings of the ACM Symposium on Applied Computing*, 2007.
- [Vit03] Vitharana, P., F. Zahedi and F. Jain: "Knowledge-Based Repository Scheme for Storing and Retrieving Business Components", *IEEE Transactions on Software Engineering*, Vol. 29, Iss. 7, 2003.
- [Voa98] Voas, J.M.: "The Challenges of Using COTS Software in Component-Based Development", *IEEE Computer*, Vol. 31, Iss. 6, 1998.
- [Vou90] Vouk, M.A.: "Back-to-Back Testing", *Information and Software Technology*, Vol. 32 (1), 1990.
- [VMo97] Bundesrepublik Deutschland: *Entwicklungsstandard für IT-Systeme des Bundes* (in german), 1997.
- [W3C04] Booth, D., H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, D. Orchard: "Web Services Architecture", *World Wide Web Consortium*, 2004.
- [Wak02] Wake, W.C.: *Extreme Programming Explored*, Addison-Wesley, 2002.
- [Wan01] Wang, N., D.C. Schmidt and C. O'Ryan: "Overview of the CORBA component model", in G.T. Heinemann and W.T. Councill (eds.): *Component-based Software Engineering: Putting the Pieces Together*, Addison-Wesley, 2001.
-

- [War03] Warmer, J. and A. Kleppe: *The Object Constraint Language: Getting Your Models Ready for MDA*, second edition, Addison-Wesley, 2003.
- [Wei01] Weinreich, R. and J. Sametinger: "Component models and component services: concepts and principles", in G.T. Heinemann and W.T. Council (eds.): *Component-based Software Engineering: Putting the Pieces Together*, Addison-Wesley, 2001.
- [Yao04] Yao, H., L. Etzkorn: "Towards a Semantic-based Approach for Software Reusable Component Classification and Retrieval", *Proceedings of the 42nd annual Southeast Regional Conference*, Huntsville, USA, 2004.
- [Ye01] Ye, Y.: "Supporting Component-Based Software Development with Active Component Repository Systems", Ph.D. Dissertation, University of Colorado, 2001.
- [YeF05] Ye, Y., G. Fischer: "Reuse-Conducive Development Environments", *Journal of Automated Software Engineering*, Vol. 12, No. 2, Kluwer, 2005.
- [Zar93] Zaremski, A.M., J.M. Wing: "Signature Matching: A Key to Reuse", *Proceedings of the Symposium on Foundations of Software Engineering*, 1993.
- [Zar95] Zaremski, A.M., J.M. Wing: "Signature Matching: A Tool for Using Software Libraries", *ACM Transactions on Software Engineering and Methodology*, Vol. 4, No. 2, 1995.
- [Zar97] Zaremski, A.M., J.M. Wing: "Specification Matching of Software Components", *ACM Transactions on Software Engineering and Methodology*, Vol. 6, No. 4, 1997.
-

LIST OF FIGURES

Figure 1.1: The reuse success chain [Fra96].....	14
Figure 2.1: Classical waterfall model [Roy70].....	22
Figure 2.2: V Model of Software Engineering.....	22
Figure 2.3: Boehm's original figure of the spiral model as in [Boe88].....	23
Figure 2.4: Graphical representation of the Rational Unified Process (from IBM website).....	24
Figure 2.5: Activities in a TDD process (source: [Amb03]).....	26
Figure 2.6: Kobra's component model [Atk02].....	32
Figure 2.7: Web service brokerage architecture.....	35
Figure 2.8: W3C's Semantic Web layer cake diagram.	37
Figure 2.9: Overview of which factors affect reuse levels and which do not.....	41
Figure 2.10: Categorization of reuse metrics and according models from [Fra96].....	44
Figure 2.11: A simple component-based software development process as proposed by [Som06].....	46
Figure 2.12: Kobra's reuse model [Atk02].....	46
Figure 2.13: Component reuse in the context of a component library as envisaged by [Ost92].....	47
Figure 2.14: Waterfall-based reuse model [Crn06].....	48
Figure 3.1: Illustration of recall and precision [Hum03].....	54
Figure 3.2: Recall versus precision curves comparing three different retrieval algorithms [Hum03].....	55
Figure 3.3: Class diagram of exemplary Stack component.....	65
Figure 3.4: Formal specification of a stack in Larch/ML as given in [Zar97].....	66
Figure 5.1: Schematic description of Extreme Harvesting process.....	96
Figure 5.2: Search space reduction.....	97
Figure 5.3: Example for a candidate result that requires an adapter.....	99
Figure 5.4: Object adapter as defined by [GoF95].....	101
Figure 5.5: Class diagram of the class adapter.....	102
Figure 5.6: A situation in which an object adapter would fail.....	103
Figure 5.7: ManagedAdapter that overcomes the problems of the GoF adapters.....	103
Figure 5.8: Sequence diagram of the testing process.....	105
Figure 5.9: Simplified class diagram of a harvested Blackjack component.....	109
Figure 5.10: System architecture for Extreme Harvesting with Merobase.....	110
Figure 5.11: Our Eclipse plugin suggesting reusable candidates based on an interface-driven search.....	111
Figure 5.12: Using Eclipse's "quick fix" function to derive a class stub from a test case.....	112

Figure 5.13: Test-driven reuse proposals by the Eclipse plugin supporting Extreme Harvesting.....	112
Figure 5.14: Harvesting system architecture.....	114
Figure 6.1: The Eclipse plugin recommends potentially useful methods for MovieTest class.....	120
Figure 6.2: The plugin presents reusable candidates that are likely to offer the required functionality.....	120
Figure 6.3: List of results that actually deliver the required functionality.....	121
Figure 6.4: Iterative component modelling process in Kobra.....	123
Figure 7.1: Initial design of Wake's running example.....	142
Figure 7.2: Screenshot of the Eclipse plugin recommending a Document class.....	143
Figure 7.3: Screenshot of the final local JUnit test run.....	145
Figure 8.1: Agora's architecture, taken from [Sea98].....	148
Figure 8.2: Process of a test-driven code search in CodeGenie.....	153
Figure 8.3: Representation of a system as a directed graph for ComponentRank [Ino05].....	155

LIST OF TABLES

Table 2.1: Potentially reusable aspects of software projects according to [Fra96].....	38
Table 3.1: Assessment of retrieval methods according to [Mil98].....	57
Table 4.1: Number of Java files indexed by search engines on the web.....	71
Table 4.2: Number of WSDL files delivered from search engines.....	72
Table 4.3: Number of WSDL files within reach at various websites (July 2005).....	74
Table 4.4: Exemplary fields contained in the Merobase index.....	79
Table 4.5: Number of components/services indexed in Merobase in summer 2007.....	80
Table 4.6: Overview of components found in version control repositories.....	81
Table 4.7: Percentage of interfaces contained in all Java files.....	81
Table 4.8: Open source licenses recognized for Java source files.....	82
Table 4.9: Number of classes using IO-Packages.....	82
Table 4.10: Overview of GUI frameworks used.....	82
Table 4.11: Distribution of special component types in the Merobase index.....	83
Table 5.1: Use cases for component search engines.....	86
Table 5.2: Fields and weights used for improved ranking within speculative and open source searches.....	88
Table 5.3: Multi-level searching for specification-based searches.....	91
Table 5.4: Possibilities for relaxed signature matches in Java.....	100
Table 5.5: Potential heuristics for resolving missing dependencies.....	108
Table 7.1: Exemplary query results from June and July 2005.....	129
Table 7.2: Exemplary query examples for more complex components.....	130
Table 7.3: Comparison of interface-based and signature-based harvesting.....	133
Table 7.4: Comparison of retrieval performance for open source searches on various search engines.....	135
Table 7.5: Comparison of code search engines performed on stateless operations.....	136
Table 7.6: Comparison of search engines with small exemplary components.....	138
Table 7.7: Comparison of retrieval techniques on stateless operations.....	139
Table 7.8: Comparison of retrieval techniques.....	141
Table 8.1: Overview of recent code search engines.....	150
Table 8.2: Overview of search engines offering library searches.....	151

APPENDIX A: TEST CASES

This appendix contains the JUnit [Bec99b] test cases, used to perform the semantic evaluation of the examples in tables 7.2, 7.3, 7.6 and 7.8, in alphabetical order.

```
public class AccountTest extends TestCase
{
    public void testDeposit()
    {
        Account a = new Account();
        a.deposit(32.33);
        assertEquals(32.33, a.getBalance(), 0.005);
    }

    public void testWithdrawal()
    {
        Account a = new Account();
        a.deposit(32.33);
        a.withdraw(20.20);
        assertEquals(12.13, a.getBalance(), 0.005);
    }
}
```

```
public class ArticleTest extends TestCase {
    public void testArticle() {
        Article art = new Article();
        art.setId(12345);
        art.setName("Navigator");
        art.setPrice(299.99);
        assertEquals(art.getId(), 12345);
        assertEquals(art.getName(), "Navigator");
        assertEquals(art.getPrice(), 299.99);
    }
}
```

```
public class BinaryTreeTest extends TestCase {
    public void testTree() throws Throwable {
        BinaryTree bt = new BinaryTree(42);
        assertTrue(bt.contains(42));
    }
}
```

```
public class CalculatorTest extends TestCase {
    private Calculator c;

    public void setUp() {
        c = new Calculator();
    }

    public void testSub() {
        Calculator c = new Calculator();
        assertEquals(-1, c.sub(4,5));
    }

    public void testAdd() {
        Calculator c = new Calculator();
        assertEquals(9, c.add(4,5));
    }

    public void testMul() {
        Calculator c = new Calculator();
        assertEquals(20, c.mul(4,5));
    }

    public void testDiv() {
        Calculator c = new Calculator();
        assertEquals(3, c.div(9,3));
    }
}
```

```
public class ComplexNumberTest extends TestCase {
    public void testAdd() {
        ComplexNumber z1 = new ComplexNumber(1.0, 1.0);
        ComplexNumber z2 = new ComplexNumber(1.0, 1.0);
        ComplexNumber z3 = z1.add(z2);
        assertEquals(2.0, z3.getRealPart());
        assertEquals(2.0, z3.getImaginaryPart());
    }
}
```

[http://www.cafeaulait.org/slides/ad2006/testdriven/Test_Driven_Development_with_JUnit.html]

```
public class CreditCardValidatorTest extends TestCase {
    public void testCardNumber() {
        CreditCardValidator ccv = new CreditCardValidator();
        assertTrue(ccv.isValid("4123456789012349"));
        assertFalse(ccv.isValid("0000"));
    }
}
```

```
public class CustomerTest extends TestCase {
    public void testCustomer() {
        Customer c = new Customer();
        c.setAddress("Baker Street 210");
    }
}
```

```
        assertEquals(c.getAddress(), "Baker Street 210");
    }
}
```

```
public class DeckTest extends TestCase {
    public void testAll() throws Throwable {
        Deck deck = new Deck();

        Card card1 = deck.dealCard();
        Card card2 = deck.dealCard();
        assertTrue(card1 != card2);

        deck.shuffle();
        Card card3 = deck.dealCard();
        assertTrue(card1 != card3 || card2 != card3);

        assertTrue( card1.toString().endsWith("Spades")
                    || card1.toString().endsWith("Clubs")
                    || card1.toString().endsWith("Hearts")
                    || card1.toString().endsWith("Diamonds") );
    }
}
```

```
public class DieTest extends TestCase {
    public void testRoll() {
        Die die = new Die( );
        for (int i = 0; i < 100; i++) {
            die.roll();
            boolean result = die.getFaceValue() > 0
                            && die.getFaceValue() <= 6;
            assertTrue("Face value out of range", result);
        }
    }

    public void testRandom() {
        Die die1 = new Die( );
        int [] count = new int [7];
        for (int i = 0; i < 12000; i++) {
            die1.roll();
            count[die1.getFaceValue()]++;
        }
        for (int i = 1; i <= 6; i++) {
            assertTrue("Non-random outcome "+i+" = "+count[i],
                      count[i] > 1900 && count[i] < 2100);
        }
    }
}
```

[<http://www.cs.vassar.edu/~cs335/Testing/DieTest.java>]

```
public class DocumentTest extends TestCase {
    public void testDocument() {
        Document d = new Document("a", "t", "y");
    }
}
```

```
        assertEquals("a", d.getAuthor());
        assertEquals("t", d.getTitle());
    }
}
```

[Wak02]

```
public class MatrixTest extends TestCase {
    Matrix matrix, matrix2, matrix3;

    public void testMatrix() throws Throwable {
        matrix = new Matrix(2, 2);
        matrix2 = new Matrix(2, 3);
        matrix3 = new Matrix(3, 2);

        matrix.set(0, 1, 42.0);
        assertEquals(matrix.get(0, 1), 42.0);

        matrix3.set(0, 0, 1.0);
        matrix3.set(0, 1, 2.0);
        matrix3.set(1, 0, 2.0);
        matrix3.set(1, 1, 3.0);
        matrix3.set(2, 0, 1.0);
        matrix3.set(2, 1, 4.0);

        matrix2.set(0, 0, 1.0);
        matrix2.set(0, 1, 2.0);
        matrix2.set(0, 2, 3.0);
        matrix2.set(1, 0, 3.0);
        matrix2.set(1, 1, 2.0);
        matrix2.set(1, 2, 1.0);

        matrix2 = matrix3.multiply(matrix2);
        assertEquals(matrix2.get(0, 0), 7.0, 0.1);
        assertEquals(matrix2.get(1, 1), 10.0, 0.1);
        assertEquals(matrix2.get(2, 1), 10.0);
        assertEquals(matrix2.get(2, 0), 13.0);
    }
}
```

```
public class MortgageCalculatorTest extends TestCase {
    public void testMortgage() {
        MortgageCalculator mc = new MortgageCalculator();
        mc.setRate(6.0);
        mc.setYears(1);
        mc.setPrincipal(100.0);
        assertEquals(8.61, mc.getMonthlyPayment(), 0.5);
    }
}
```

```
public class MovieTest extends TestCase {
    public void testTitleRetrieval() {
        Movie movie = new Movie("Star Wars", 0);
        assertEquals(movie.getTitle(), "Star Wars");
    }
}
```

```
    }  
}
```

[Fow99]

```
public class ShoppingCartTest extends TestCase {  
    private ShoppingCart cart;  
    private Product book1;  
  
    protected void setUp() {  
        cart = new ShoppingCart();  
        book1 = new Product("Pragmatic Unit Testing", 29.95);  
        cart.addItem(book1);  
    }  
  
    public void testEmpty() {  
        cart.empty();  
        assertEquals(0, cart.getItemCount());  
    }  
  
    public void testAddItem() {  
        Product book2 = new Product("Pragmatic Project Automation", 29.95);  
        cart.addItem(book2);  
        double expectedBalance = book1.getPrice() + book2.getPrice();  
  
        assertEquals(expectedBalance, cart.getBalance(), 0.0);  
        assertEquals(2, cart.getItemCount());  
    }  
  
    public void testRemoveItem() {  
        cart.removeItem(book1);  
        assertEquals(0, cart.getItemCount());  
    }  
}
```

[<http://clarkware.com/articles/JUnitPrimer.html>]

```
public class SortTest extends TestCase {  
    public void testSort() {  
        int[] odd = {8, 2, 3, 1, 24, 13, 5, 4};  
        int[] sorted = {1, 2, 3, 4, 5, 8, 13, 24};  
        Sort s = new Sort();  
        s.quickSort(odd);  
        for (int i=0;i<odd.length;i++)  
            assertEquals(odd[i], sorted[i]);  
    }  
}
```

```
public class SpreadsheetTest extends TestCase {  
    public void testCellReference() {  
        Spreadsheet sheet = new Spreadsheet();  
        sheet.put("A1", "5");  
        sheet.put("A2", "=A1");  
        assertEquals("5", sheet.get("A2"));  
    }  
}
```

```
    }

    public void testCellChangePropagates() {
        Spreadsheet sheet = new Spreadsheet();
        sheet.put("A1", "5");
        sheet.put("A2", "=A1");
        sheet.put("A1", "10");
        assertEquals("10", sheet.get("A2"));
    }

    public void testFormulaCalculation() {
        Spreadsheet sheet = new Spreadsheet();
        sheet.put("A1", "5");
        sheet.put("A2", "2");
        sheet.put("B1", "=A1*(A1-A2)+A2/3");
        assertEquals("15", sheet.get("B1"));
    }
}
```

[<http://today.java.net/lpt/a/69>]

```
public class StackTest extends TestCase {
    public void testAll() throws Throwable {
        Stack stack = new Stack();
        stack.push((Object) "Lassie");
        stack.push((Object) "Fury");
        stack.pop();
        stack.push((Object) "Flipper");
        stack.push((Object) "Fury");

        assertEquals(stack.pop(), (Object) "Fury");
        assertEquals(stack.pop(), (Object) "Flipper");
        assertEquals(stack.pop(), (Object) "Lassie");
    }
}
```

It's not the pace of life that concerns me, it's the sudden stop at the end.

-- Unknown
