A Dependency-Aware, Context-Independent Code Search Infrastructure

Inauguraldissertation

zur Erlangung des akademischen Grades

eines Doktors der Naturwissenschaften

der Universität Mannheim

vorgelegt von

Diplom-Informatiker Marcus Schumacher

aus Heidelberg

Mannheim, 2019

Dekan:	Dr. Bernd Lübcke, Universität Mannheim
Referent:	Prof. Dr. Colin Atkinson, Universität Mannheim
Korreferent:	Prof. Dr. Ralf H. Reussner, Karlsruher Institut für Technologie (KIT)

Tag der mündlichen Prüfung: 18.07.2019

Abstract

Over the last decade many code search engines and recommendation systems have been developed, both in academia and industry, to try to improve the component discovery step in the software reuse process. Key examples include Krugle, Koders, Portfolio, Merobase, Sourcerer, Strathcona and SENTRE. However, the recall and precision of this current generation of code search tools are limited by their inability to cope effectively with the structural dependencies between code units. This lack of “dependency awareness” manifests itself in three main ways. First, it limits the kinds of search queries that users can define and thus the precision and local recall of dependency aware searches (giving rise to large numbers of false positives and false negatives). Second, it reduces the global recall of the component harvesting process by limiting the range of dependency-containing software components that can be used to populate the search repository. Third, it significantly reduces the performance of the retrieval process for dependency-aware searches.

This thesis lays the foundation for a new generation of dependency-aware code search engines that addresses these problems by designing and prototyping a new kind of software search platform. Inspired by the Merobase code search engine, this platform contains three main innovations - an enhanced, dependency aware query language which allows traditional Merobase interface-based searches to be extended with dependency requirements, a new “context independent” crawling infrastructure which can recognize dependencies between code units even when their context (e.g. project) is unknown, and a new graph-based database integrated with a full-text search engine and optimized to store code modules and their dependencies efficiently. After describing the background to, and state-of-the-art in, the field of code search engines and information retrieval the thesis motivates the aforementioned innovations and explains how they are realized in the DAISI (Dependency-Aware, context-Independent code Search Infrastructure) prototype using Lucene and Neo4J. DAISI is then used to demonstrate the advantages of the developed technology in a range of examples.

Zusammenfassung

Im letzten Jahrzehnt wurden sowohl im akademischen als auch im industriellen Bereich zahlreiche Code-Suchmaschinen und so genannte Recommendation-Systeme entwickelt, um den ersten Schritt im Prozess der Wiederverwendung von Softwarekomponenten zu verbessern, die Suche nach passenden Komponenten. Bedeutende Beispiele der letzen Jahre waren oder sind Krugle, Koders, Portfolio, Merobase, Sourcerer, Strathcona oder SENTRE. Allerdings ist die Trefferquote und die Genauigkeit dieser aktuellen Generation an Programmen für die Code-Suche gewissermaßen begrenzt, da sie nur bedingt in der Lage sind strukturelle Abhängigkeiten zwischen verschiedenen Code-Einheiten effektiv dar zu stellen. Dieser Mangel an “Abhängigkeitsbewusstsein” findet sich dabei in drei Hauptaspekten. Erstens ist die Art wie Benutzer Suchanfragen an das System definieren können, und damit die Genauigkeit und die lokale Trefferquote abhängigkeitsbewusster Suchen, eingeschränkt (was zu einer großen Anzahl von false-positive und false-negative Ergebnissen führt). Zweitens ist die globale Trefferquote des Komponenten-Harvesting-Prozesses verringert, da die Möglichkeit Softwarekomponenten, die Abhängigkeiten enthalten, in einem der Suche zu Grunde liegenden Such-Repository abzubilden begrenzt ist. Drittens ist die Effizienz des Prozesses des Information Retrieval für abhängigkeitsbezogene Suchen signifikant reduziert.

Diese Dissertation legt den Grundstein für eine neue Generation abhängigkeitsbezogener Code-Suchmaschinen, bei der diese Probleme und Einschränkungen durch den Entwurf und das Prototyping einer neuen Art einer Suchplattform für Software gelöst werden. Inspiriert von der Merobase-Codesuchmaschine werden hier drei Hauptinnovationen präsentiert: eine erweiterte, abhängigkeitsbezogene Abfragesprache, mit der herkömmliche, auf der Merobase basierende Suchanfragen, um Abhängigkeitsanforderungen erweitert werden können. Eine neue, "kontextunabhängige" Crawling-Infrastruktur, die Abhängigkeiten zwischen Codeeinheiten erkennen kann. Und die Integration einer Graphen-Datenbank in eine Volltextsuchmaschine die auf eine effiziente Speicherung von Codemodulen und deren Abhängigkeiten optimiert ist. Nach den Grundlagen und den aktuellsten Techniken auf dem Gebiet der Codesuchmaschinen im Bereich des Information Retrieval motiviert die Dissertation die oben genannten Innovationen und erläutert, wie diese innerhalb der DAISI (Dependency-Aware, Context Independent Code Search) auf Basis von Lucene und Neo4J umgesetzt werden. Anhand der DAISI werden auch die Vorteile der entwickelten Technologie anhand einer Reihe von Beispielen demonstriert.

Acknowledgment

After many years of intensive work, this Acknowledgment represents the culmination of my dissertation project. The work was intensive but also rewarding, not only due to the many things I was able to learn exploring my research topic but also due to the many interesting people I was able to work and interact with. It was always a pleasure being at the chair, and over the years many of my colleagues have become friends.

First and foremost I would like to thank my supervisor, Colin Atkinson, for giving me the chance to research on this stimulating topic and for always having an open ear for questions and productive discussions. The atmosphere you have created at your chair helped me in many ways, and provided the freedom needed for creative thinking. Without this, many of my colleagues and I would not have been able to come up with such innovative solutions to the problems we tackled.

Second, I would like to thank all the colleagues who have accompanied me all these years, especially Oliver Erlenkämper, Werner Janjic, Ralph Gerbig, Thomas Schulze and Marcus Kessel. Thank you for your encouragement and all the discussions in the coffee corner where a lot of ideas were born (about all manner of things including our research topics).

And last but not least, a big thanks to my family - my wife Kerstin, who has loved, supported and encouraged me all these years, and my sons Fabian and Florian, who sometimes had to put up with playtime interruptions when their father had to check on his research prototype or implement a new idea. It has been a struggle to get this dissertations finished, but sometimes the best research ideas came up when building turrets or houses. I therefore dedicate this work to my family.

Contents

List of Figures

List of Tables

1. Introduction

Google can bring you back 100,000 answers. A librarian can bring you back the right one.

– Neil Gaiman –

Today software permeates almost every part of our lives and environment, whether it be as programs within computers, applications on smartphones, embedded controllers within consumer goods or artificial intelligence within autonomously driving cars. Software lies at the heart of all modern, "smart" products. However, the development of software is still a tremendously costly process. Although there have been many changes in software engineering approaches over the last 50-60 years, with waterfall processes gradually giving way to agile processes [Som01], software development still primarily revolves around the notion of writing code from scratch.

The idea of systematically building new applications from pre-existing components was first promoted in the 1960's to increase software quality and raise productivity [McI68] [Moh+04]. However, today reuse is essentially only practised in ad-hoc, opportunistic ways by individual developers who happen to be aware of existing software that could fulfil their needs [LM89] [HW07]. Large scale, systematic reuse that mirrors a component-based assembly like in other industries, e.g. the automotive industry, where almost all new cars have standard, reusable components installed (e.g. radio, air conditioning, fuel pumps etc.), is in software engineering still a long way off, although the benefits of prefabricated component assembly in software engineering are potentially just as dramatic. For example, Lime found that the defect density in software systems built from existing components was half that of systems developed from scratch through normal processes [Lim94].

However, systematically supporting software reuse is difficult because there are so many different forms of components and ways of reusing them. A "component" can be as small as a block of code or as large as a complete subsystem or framework, and component reuse can take many different forms, depending on how many components are involved and the reuser's degree of knowledge about a component's internal realization. In terms of quantity, a developer may sometimes only wish to reuse one independent component and on other occasions a developer may wish to reuse several interconnected components. In terms of knowledge, a developer may sometimes want to reuse a component "as-is" in a black box way, without any knowledge about how it works internally, and on other occasions a developer may wish to reuse a component in a white box way by modifying it for the task in hand or simply learning from the way it is implemented [Sim+11].

Those differences have an impact on all phases of the reuse process. However they present by far the biggest challenge for the first and arguably most important step which is to find suitable reuse candidates in the first place. Probably the most effective way of boosting software reuse in software engineering is to provide easier and more reliable ways for developers to find components to support all the different kinds of components and reuse forms. Over the last few years there have been many attempts to improve the component discovery step in the software reuse process, including the development of code search engines like Krugle [Kru13], Koders, Portfolio

[McM+11], Merobase [Jan+13] or Searchco.de [1] and plug-in recommendation tools such as Strathcona [HM05] or SENTRE [Jan14]. However, the current generation of software search engines or recommendation systems only supports a few of the aforementioned component types and reuse scenarios. In fact, most developers still use text-based searches on Google, or similar web search engines, to find components to use "as-is" or to find reference examples they can use in their own applications [Sim+11]. Many important cases are only supported in a very rudimentary way or not at all. But as the quote by Gaiman at the beginning hinted "Google can bring you back 100,000 answers. A librarian can bring you back the right one.", today's software search engines can return many results to almost every query, but they are frequently not the right ones. To improve the precision of software search engines it is necessary to make them aware of the structure of software components as well as of the "text" elements they contain.

1.1 Dependency Awareness

As an example of the kind of challenges faced by developers when trying to use today's code search engines to support reuse, suppose a developer is responsible for building a customer management (sub-)system based on the core classes and relationships shown in figure 1.1. The central class in this system is the " *Customer-Management* " class which is the entry point for looking up customers, updating customers and creating new ones. This class is therefore responsible for managing multiple instances of the class "*Customer*". At the beginning of the development process, the detailed properties of these classes are largely unknown, but it is clear that the *Customer* class must at least contain attributes to store customer names and addresses.

[1] http://www.searchco.de

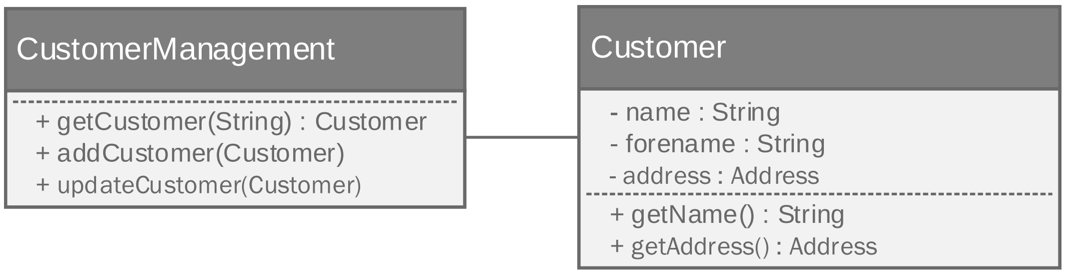


Figure 1.1: Simple CustomerManagement System

A developer who is open to reusing existing code to build this subsystem might start by entering a query of the form “java customermanagement customer getname” in a general purpose search engine such as Google and as for example shown by Sim et al. [Sim+11], most developers turn to such search engines to find code. At the time of writing, entering this query into Google returned only one remotely relevant result at position two – a single Java class called *CustomerManagement* hosted at GitHub. The class *Customer* is completely missing in this result. Entering this query into Koders or Krugle, the two most popular and well known code search engines, returned no results at all. The problem is the sensitivity of these search engines to the exact keywords supplied in the query. If the identifiers in a component stored in the repository deviate in the slightest way from the keywords in the query, these search engines are unable to detect a match.

To provide better results, Krugle and Koders both provide special prefixes to identify what role specific keywords should play in software. For example, it is possible to obtain relevant results for this scenario from Krugle by changing the query to “customermanagement functiondef:getCustomer”, and in the case of Koders a much more effective query is “cdef:customermanagement mdef:getCustomers”. This specifies that, to be included in the result set, a class must either have the name *CustomerManagement* or a method called *getCustomers*, or both. The inability of the first example query to return any results from Koders and Krugle suggest they do not have any classes called *CustomerManagement* in their index. Therefore, in the case of Krugle it was also necessary to change the individual query to "customer functiondef:getCustomer" where the class name was changed from *customermanagement* to *customer*, and in the case of Koders to "cdef:customerservice mdef:getCustomers" to search for a *customerservice* instead of the *customermanagement*.

Of course, developers are usually interested in the functionality offered by components, not their exact name. However, unlike Google major code search engines like Koders and Krugle are unable to take similar names into account when searching for components. Additionally, in both cases the given query was unable to convey the requirement for the *getName*-method. General purpose search engines like Google provide no special prefixes for code and are thus unable to recognize the types or names of any input- / output-parameters of methods. Google simply matches keywords in the query to identifiers in the source code, regardless of what they mean, but applies sophisticated name similarity detection techniques. In summary, Google almost always returns a high number of results, but the vast majority of them are irrelevant (low precision), while mainstream code search engines such as Krugle and Koders often return few if any results (low recall).

The new generation of academic search engines developed over the last couple of decades such as Koders, Portfolio, Merobase or Searchco.de have tried to address this problem by introducing new ways of formulating queries. For example, Merobase introduced interface-based searches in which queries essentially describe the signature information wrapped up in interface specifications. For example the query -

```
Customer(getName():String; getId():String; setName(String))
```

will search for components (e.g. classes) called *Customer* offering methods *getName* and *getId* which return a String value and *setName* which accepts a string parameter. This significantly increases the precision of the results compared to the aforementioned search engines which simply look for the union of the listed features. However, the results are still very sensitive to the identifiers chosen in the query. Merobase therefore offers a variant of interface based search, called signature based search, which only specifies the signatures of the required methods, without specifying the identifiers. This, of course, has the effect of increasing the recall again but at the expense of precision. However, while these more advanced query specification capabilities allow precision and recall to be improved, especially when combined with systematic query reformulation techniques, they still focus on individual components.

The basic problem with existing code search engines is their lack of "awareness" of the typical kinds of dependencies that exist between code elements. This is a fundamental problem because in the object-oriented programming languages primarily used to write software today, functionality is almost always distributed over several classes. Making search engines, and the queries that drive them, "dependency aware" would allow them to consider the content of several related classes rather than one individual class when establishing their relevance for the user.

In case of the Customer Management example, not only would it be possible to search for a *CustomerManagement* class which has some kind of relationship to a *Customer* class, it would also be possible to specify the precise kind of relationship and properties of the *Customer* class, like *name* or *address* attributes. Additionally, the immediate environment of each class could be examined so that methods that are not contained by the class itself, but are defined in other classes (e.g. superclasses) can be taken into account. Dependency-aware search technology therefore has the potential to further enhance the precision and recall of searches. It can enhance the former by allowing users to specify precisely what elements they expect, components to be composed of and what relationships should exist between them. It can enhance the latter by allowing components to be considered as candidates that do not provide all of the desired functionality themselves but do so with the help of other dependent components.

The first step towards exploiting dependency awareness in code searches was made by the search engine Portfolio [McM+11] which considers the process flow of method calls in related classes, and SENTRE, which loads all dependencies after a component is selected [Jan14]. However, they exploit dependency awareness in only very limited and narrowly focused ways. At the present time there is no code search engine which allows the full range of dependencies supported by modern object-oriented programming languages to be explicitly considered in the formulation and execution of code searches.

1.2 Context-Independent Harvesting

As soon as the focus of searches extends beyond individual code elements to collections of elements, the question of how these elements are correlated and harvested becomes a major question. More specifically, the problem of how to examine whether elements belong together has to be addressed. The simplest and most straightforward approach, if possible, is to harvest all elements from a given project or package (e.g. jar file) because they are related to each other in a given "context" and were usually written by the same development team. The most common and straightforward way of harvesting elements and their dependency relationships is to collect them from online project repositories such as Github or Sourceforge, which are typically configured by dependency management tools like Maven, Ivy or Gradle. They capture dependencies in configuration files, along with all the inter-related code elements within projects. However, such configuration files can often get lost or be corrupted, or for some reason another code element may become unavailable (e.g. due to versioning problems). This makes it impossible for crawlers that rely on full context information to harvest such code. Also, a great deal of reusable code is available as individual elements and code snippets in internet forums like Stackoverflow [2]. Moreover, these code snippets are often of very high quality [SH13a] because they address concrete problems that are discussed and explained by a large number of developers. A context-independent dependency resolution mechanism which can inter-connect elements harvested from different contexts would therefore significantly enhance a dependency aware search engine by allowing it to (a) harvest more interrelated code elements (with an awareness of the dependencies between them), and (b) be more robust against events that break the integrity of the index, such as the movement of a project to another server.

Since context-dependent harvesting is much easier than context-independent harvesting we assume that any search engine that does the latter will also do the former. In other words, we assume that context-independent harvesting subsumes context-dependent harvesting. Conceptually, if a repository over which searches are performed is regarded as complete, e.g. consists of all existing software components which can be found in the global internet, rather than a subset, created by a search engine's crawler, a context-independent harvesting mechanism essentially increases the global recall of a search engine since it allows more component candidates to be considered.

The benefits of such context-independent harvesting can be directly seen in the example above. To reuse the result returned by Google from the Github project, which consists of only one class, *CustomerManagement*, a developer either has to implement the second class *Customer* by himself or has to perform a second search. Even if the implementation of the *Customer* class may not be that difficult, a context-independent harvesting process would be able, in the case of a complete index, to find another *Customer* class containing the necessary structure and information and

[2]http://www.stackoverflow.com

relate it to the *CustomerManagement* class.

1.3 Research Goals

To address the aforementioned issues this thesis presents the foundations for a new generation of "dependency-aware" search engines which can be populated through "context-sensitive" harvesting, and demonstrates their effectiveness by means of a prototype implementation. In addition, the new technologies created for this purpose were developed to fulfil the following sub-goals:

1. **seamless extension**: seamlessly extend the capabilities of the current generation of code search engines.
2. **language independence**: ensure compatibility with all mainstream object-oriented programming languages and software repositories.
3. **scalable**: provide reasonable performance (i.e. response time) even for very large scale repositories.

The general hypothesis behind this work is, therefore, that this new search engine technology, which we refer to as DAISI (Dependency-Aware, context-Independent code Search Infrastructure), will increase the precision and recall of code search engines, both in terms of "local recall" relative to the harvested code repository, and in terms of the "global recall" relative to the conceptually available components in the Internet. The particular search engine chosen as the baseline for the research was the Merobase search engine also developed at the Chair of Software Engineering at the University of Mannheim. Wherever possible, the new capabilities were added to the existing features of Merobase (e.g. the query language and document indexing structure) in an as natural and simple way as possible. The particular language chosen as the focus of the prototype implementation was the Java programming language.

When it comes to storing information about relationships, the text-based document indexing approaches used to implement the current generation of code search engines are not scalable for a stand-alone solution. Relational databases are also unsuitable to cover the requirements of solid relationship-based dependency discovery because of the large number of "joins" that are often required to resolve relationships [HAS13]. This thesis therefore explores the novel approach of using a combination of traditional text-based indexes and a graph database to store the searchable content.

In order to achieve the second sub-goal of being language independent, the document schemata and graph database nodes need to be correlated and language agnostic (i.e able to cope with all major object-oriented language concepts). A major contribution of the work presented in this thesis is therefore the definition of a language-independent metamodel which captures the key information needed to support dependency-awareness in an efficient and level-agnostic way.

Finally, in order to support context-independent harvesting it was necessary to design a two-phase parsing process, and a generic parsing approach which on the one hand supports the abstract dependency representation approach defined in the language-agnostic metamodel and on the other hand supports the straightforward implementation of language specific parsers and analysers.

1.3.1 Hypotheses

The research goals described in the previous section essentially postulate the validity of the following concrete hypotheses -

Hypothesis 1

It is possible to build a scalable, context-independent, dependency-aware, code search engine populated through context-independent harvesting.

The premise behind this hypothesis is that it is possible to build a code search engine that demonstrates the desired properties of language agnosticism and dependency awareness that is "practical" in the sense that it is (a) able to harvest and index a "meaningful" collection of components (in a context independent way) and (b) able to return results to dependency-aware searches over a "large" repository within a "reasonable" amount of time. The validity of this hypothesis is not self-evident since no code search engine with these properties has yet been built. Although the terms "meaningful", "large" and "reasonable" are not quantified precisely and thus open to interpretation, they can nevertheless be concretely measured against the capabilities of existing code search engines

Hypothesis 2

A dependency-aware code search engine of the kind referred to in Hypothesis 1, which allows users to express the dependency relationships they desire between code elements when defining queries, can enhance the precision of search results.

The premise behind this hypothesis is that if (a) users can precisely express what software structures they are looking for, containing the precise kinds of dependencies they need, and (b) a search engine can effectively deliver results from a large repository that stores these kinds of dependencies, the number of unsuitable (i.e. undesired) software structures returned in search result sets will be reduced. Technically, this corresponds to an increase in the precision of the search engine measured against the “true” requirements of the user. This increase in precision is only expected for dependency-containing queries. There is no claim that the precision of normal (i.e. dependency-free) queries is increased, although this might be a side-effect of the graph-based implementation approach used to support dependency-aware searches.

Hypothesis 3

A dependency-aware code search engine of the kind referred to in Hypothesis 1 can enhance the (local) recall of search results.

This is the “flip-side” of hypothesis 2, but related to recall rather than precision. The premise behind this hypothesis is that if (a) users can precisely express what software structures they are looking for, containing the precise kinds of dependencies they need, and (b) a search engine can effectively deliver results from a large repository that contains these kinds of dependencies, the number of suitable (i.e. desired) software structures included in the result set is increased. Technically, this corresponds to an increase in the recall of the search engine measured against the “true” set of suitable components in the repository. Again, this increase in precision is only expected for dependency-containing queries, and there is no claim that the recall of normal (i.e. dependency-free) queries is increased, although this might be a side-effect of the graph-based implementation approach used to support dependency-aware searches.

Hypothesis 4

A dependency-aware code search engine of the kind referred to in Hypothesis 1, populated by a context-independent harvesting approach, can enhance the (global) recall of search results.

Hypothesis 3 refers to an expected increase in the recall of dependency-aware searches measured in terms of the components that are contained in the search engine's repository. Thus, the "true" set of suitable components against which recall is judged is the set of query-matching components contained in the repository. In this thesis we refer to this form of recall as "local recall". However, since we are focussed on "open" code search engines which harvest software from "open" source software repositories and forums publicly accessible anywhere on the Internet, the "virtual" repository against which users conceptually issue their search queries is the complete set of components that are in principle harvestable from the open internet. In other words, for users of online, open search engines, the "true" set of suitable components against which recall is judged is, conceptually at least, the set of query-matching components in the openly harvestable Internet. We refer to this form of recall as "global" recall. The key difference between local recall and global recall is that global recall takes into account the effectiveness of the component harvesting/indexing technology as well as the component retrieval technology, while local recall only takes the former into account. Since we believe the context-independent harvesting approach will be able to find and process more components, we believe it will contribute towards an increase in the global recall of the search engine.

1.4 Thesis organization

The remainder of this thesis is organised in the following way. Chapters two, three and four provide the detailed background needed to understand the problem domain tackled by the thesis and the nature of the proposed solution. Here, chapter two provides a general overview of information retrieval concepts and technologies, while chapter three describes the specific problems that are encountered when trying to apply them to software. Chapter four provides a detailed overview of the most influential code search engines that have been developed to date, both in industry and academia.

With the background established, the next three chapters present the new approaches and technologies proposed in thesis and describes the prototype code search engine, DAISI, developed to demonstrate and test them. Chapter five begins by describing the metamodels that were developed to describe the underlying data structure used to support the new technology. These metamodels exist at various levels of abstraction so that on the one hand they are general enough to be applied to all kinds of programming languages and software environments, and on the other hand are concrete enough to support efficient application to a tangible example, in this case the Java programming language. Chapter six continues by presenting the technology behind the context-independent harvesting aspect of the new technology. This includes a detailed explanation of how online software component repositories are crawled and analysed, and how the harvested software is stored in a Neo4J graph database alongside multiple Lucene NLP (Natural Language Processing) indexes. Chapter seven continues the detail exposition of the developed technology by describing the dependency-aware search aspect of the approach. This includes a description of the new dependency-aware query language, DAQL, as well as an explanation of the various new kinds of searches and features that can be carried out by users.

The final two chapters round off the thesis by discussing the effectiveness and potential impact of the new search engine technology developed in the thesis. Chapter eight presents a concrete evaluation of the developed prototype search engine using increasingly complex search scenarios based on the running example used in the thesis, and demonstrates that all the hypotheses outlined in the previous section are valid, at least for this example. Finally, chapter nine concludes by summarising the main contributions of the thesis and discussing the potential impact and possible future enhancements of the developed technology.

2. General Information Retrieval Concepts

The need for information consists of the process of perceiving a difference between an ideal state of knowledge and the actual state of knowledge

– Lidwien van de Wijngaert –

The ability to build new software applications by assembling reusable components, rather than developing code from scratch, has been the dream of software engineers for decades. However, the industry is still a long way from realizing this vision in mainstream software engineering. The vision was first framed in 1986 by McIllroy, but serious research on the topic was kick-started by Kruger in 1992 with his definition “Software reuse is the process of creating software systems from existing software rather than building software systems from scratch” [Kru92]. Since then many researchers have worked on trying to turn this vision into practical reality. Although good progress has been made, significant parts of the reuse process still present serious obstacles. One of the biggest challenges is the general problem of finding suitable components to reuse in the first place. As recently as 2001, Sommerville stated that the search for software components is one of the most critical elements of an effective reuse program [Som01]. Moreover, Singer et al. in 1997 [Sin+97] and Murphey in 2006 [MKF06] characterized the search for code as one of the most critical stages in the software reuse process.

The search for software components differs in some important aspects to searches for other kinds of artefacts. This is because of searches for software components are essentially searches for "behaviour" or "functionality" whereas searches for most other kinds of artefacts (physical or informational) are essentially searches for "properties". Since general search engine technology is ultimately driven by property matching, it is much easier to find suitable components based on properties rather than behaviour. However, in the case of software components, there are so many ways in which functionality can be mapped to code, it is very difficult to judge what concrete properties a search engine should be looking for when trying to identify suitable components. The many options for implementing a given piece of functionality arise not only because of the availability of many different programming languages, but also because there are usually numerous architectures and algorithms that can be used to solve a particular problem. Moreover, the strings used to name the classes and methods within a program are entirely at the discretion of the developer. Precisely these variations complicate the search for software components and have challenged researchers for many years.

Key questions for researchers include the structure and form of a query language for software components and the structure of the index or the database that stores the repository of components to be searched. Effective answers to these questions must take into account the many idiosyncrasies of software implementation technologies and the many different ways software engineers can map a given set of requirements into corresponding implementations. To date, most research has focused on handling the naming variations that can exist in a domain, to ensure that identifiers used in component implementations somehow match those used to formulate the search query.

Initial attempts to tackle these problems were rather simple, such as the Japanese "factory approach" for software development [Cus89] which various Japanese companies introduced to try to support a reuse program [Mat84]. These mechanisms and approaches have been continuously improved over time, however, so that today naming differences and implementation variations can be handled reasonable well. Therefore, information retrieval techniques still form the backbone of software search engines. This chapter provides an overview of these techniques and evaluates their relevance for software search engines today.

2.1 Recall and Precision

Information retrieval (IR), a term introduced in the late 1940's by Calvin Mooers [Moo50], has its origins in library science. The era of computer-supported retrieval, which can be traced back to the late 1940's [Cle91] [Lid05], arose from the need to archive the rapidly growing number of newspaper articles and scientific papers caused by the emergence of computing technologies. However, the first widely-accepted definition of information retrieval was made nearly 20 years later by Gerald Salton [Sal68]:

" Information retrieval is a field concerned with the structure, analysis, organization, storage, searching, and retrieval of information."

Given the limited hardware and programming capabilities at that time and the original focus on physical books, scientific articles, newspapers and later e-mails, the initial types of searches supported were very simple. For many years it was only possible to specify the author, the title or some specific key word characterizing the desired article. However, with the invention of the World Wide Web (WWW), these early technologies were no longer sufficient and a new generation of IR technologies was required. Among other things this new generation had to cope with significantly more information. For example, in the 1960's the data stored in a new IR system was only about 1.5 megabytes, whereas today billions of online information artefacts are available which must be analysed and made searchable.

Nevertheless, the basic concepts used to measure the effectiveness of search engines are still the same. One of most fundamental is "relevance" [CMS10] which expresses how well the items returned by an information retrieval system match the requirements or goals of the users. Basically, a document can be seen as relevant if it contains the information or the properties the user of the search engine is looking for.

Relevance can actually be split into two categories. One is the *topical* relevance and the other the *user* relevance. The difference between these two relevance categories lies in the information the retrieved documents contain. In the case of topical relevance, the search results must fit only to the topic of the search query, whereas in the case of user relevance additionally conditions and criteria must be satisfied as well. For example, one of the results returned by a search for the German word "Fussballweltmeister" (Soccer world champion) on one of the biggest existing search engines is "Italy". From one point of view this may be correct and relevant since Italy won the world cup four times in the past. However, Italy is not the current world champion and is thus not a relevant result for a user searching for the last team which won the competition. Exactly these kinds of factors contribute in subtle ways to the user relevance of search results [CMS10].

These different forms of relevance are important in the area of software search engines. For example in the previous chapter, a search for components which use a class called *Customer* could also deliver results which only have the word *Customer* somewhere in the comments, but are not actually using a class with this name. From a topical relevance point of view, this is a relevant result, but in terms of the behaviour required by the user it is completely irrelevant. Relevance, by itself, is not a sufficiently accurate measure, therefore. The quality of search results strongly depends on the expectations of the user. In 1961, Cyril Cleverdon therefore introduced the metrics “precision” and “recall” to better quantify the effectiveness of IR systems and to allow their ranking algorithms to be validated and compared [Cle61].

Precision: the fraction of the documents retrieved from the search base that are relevant to the user’s information need.

Recall: the fraction of the documents in the search based that are relevant to the query that are successfully retrieved.

Both metrics are still widely used today to evaluate new IR approaches. However, the accuracy of the second requires knowledge of the total number of relevant documents in the search base, which is very difficult to establish in practice, especially given the size of search bases today (e.g. billions of documents from the global Internet) and the rapid rate at which they change [GP99]. To address this problem, there are several models for describing the conceptual space a search engine operates in. Among the oldest IR models are the Boolean Retrieval Model and the Vector Space Model [B+99]. These models, which are both still used today, are the foundation of many subsequently developed approaches. As shown in figure 2.1, at the core of every IR model is a quadruple $[D, Q, F, R(q_i, d_j)]$, where D is a logical view on the documents in the collection, Q is a logical view of the user queries, F is the framework for representing the documents and queries and $R(q_i, d_j)$ is the ranking function.

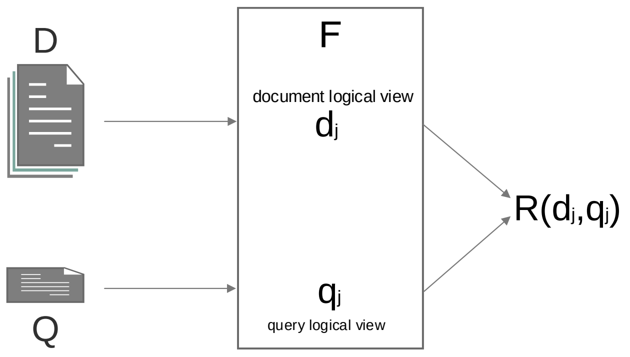


Figure 2.1: IR Model

2.2 Boolean Retrieval

The Boolean Retrieval Model, sometimes also known as exact-match retrieval, was used in some of the first IR systems. As its name suggest, the basic idea is to include only exact matches in the result list. In other words, results are only included in the result list if they contain all of the keywords specified in the search query and satisfy all the Boolean constraints it implies. The term “implies” is used here because a query can be a complex expression whose parts are connected by logical operators. In the beginning only the basic and well know logical operators *AND*, *OR* and *NOT* where supported, so a query could take the form $q = q_a \wedge (q_b \vee q_c) \neg q_d$, where q_i can be also sub-expressions of the query q.

While it is relatively easy for a user to formulate such a request, formulating queries in the Boolean Retrieval Model has some weaknesses. Since it assumes that relevance is also binary and all “true” results included in the result set are basically equivalent to one another, the approach provides no inherent ranking mechanism. Apart from some ordering strategies (such as sorting by date) therefore, the user can often receive quite a lot of irrelevant results, even though the overall precision is high. It can therefore be quite a complex task to formulate a search query to get results with a high degree of user relevance. The core problem here is that the Boolean Retrieval Model does not consider how often or in what kind of context the key words are present in the document. This weakness still persists in some of the more recent enhancements to this model, such as the introduction of the wild-card character or mechanisms to search with regular expressions. Other extensions, however, have integrated a ranking mechanism, mostly by combining the Boolean Retrieval Model with the Vector Space Model [SFW83], presented in the next section.

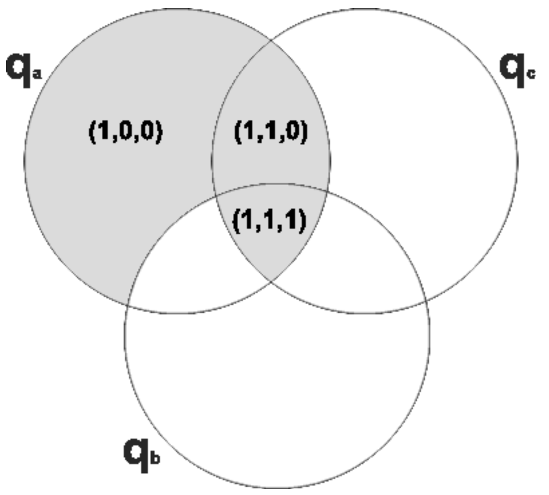


Figure 2.2: Boolean Retrieval Model Query conjunctive components

2.3 Vector Space Model

The Vector Space Model was one of the earliest IR retrieval approaches (alongside the Boolean model) and became one of the main focus of IR research in the 60s and 70s [B+99] because it offers some key advantages – namely, ranking, term weighting and relevance feedback. It achieves this by regarding all documents and queries as vectors in a t-dimensional vector space, where t is the number of index terms within a document, like words or sentences. The vector itself is composed of index terms $D_i = (d_{i1}, d_{i2}, \ldots, d_{it})$, where d_{ij} represents the weight of the j-th term. The vectors of all documents are then combined into a matrix which constitutes the index, where each row stands for a document and each column for the weightings of the different terms within that document.

$$\begin{array}{ccccc}
 & Term_1 & Term_2 & \dots & Term_t \\
Doc_1 & d_{11} & d_{12} & \dots & d_{1t} \\
Doc_2 & d_{21} & d_{22} & \dots & d_{2t} \\
\vdots & \vdots & & & \\
Doc_n & d_{n1} & d_{n2} & \dots & d_{nt}
\end{array}$$

Search queries in the Vector Space Model are represented in a similar way as vectors of the form $Q = (q_1, q_2, \dots, q_3)$ where q_j again represents the weighting, but in this context the weighting of the j-th term within the query. Since a query vector is always as long as a column in the index matrix, the results can be determined with the help of a distance calculation and a determination of its resemblance to the matrix. Over the years, cosine correlation has proved to be the most effective resemblance and distance measure.

The biggest disadvantage of the Vector Space Model is the number of dimensions t. The approach simply does not scale up to the billions of documents that are today harvestable over the Internet because the dimension t simply become too large to manage and the time taken to calculate distances becomes to long. This problem was recognized in early uses of this approach in terms of the sizes of vectors for certain books. Vector normalization therefore became a common practice to ensure all vectors have a uniform length.

A big advantage of the Vector Space Model over the Boolean Retrieval Model is that the distance calculation yields a non-binary measure of a component's relevance which can be used as the basis for ranking. However, this also means that every time the search query is changed the distance measures have to be re-calculated [CMS10]. Nevertheless, this inherent ranking capability laid the foundations for new approaches addressing the relevance problem and improving ranking still further.

2.4 Set-based Model

Like many other IR models, the Set-based Model introduced in 2002 has its origins in the Vector Space Model [Pôs+02]. One of the main improvements introduced in this model is to combine rules from the association theory [AIS93] with the vectorial ranking mechanism. This allows it to take inter-dependencies between different index terms into account and include documents in the result list which do not actually contain any of the specified keywords in the search query. Instead of storing the terms in documents directly in an index, this is achieved by referencing "termsets" where each termset contains all correlated terms that express the same meaning in different words. This approach was the first mechanism for relating different documents to one another, where the relationship between two documents is determined by the degree of similarity of the referenced termset. Instead of the general termsets the Set-based Model makes use of so called closed-termsets. By reducing the computational complexity and the amount of data to be stored without losing information, closed-termsets have two important advantages over maximal termsets or frequent termsets [Pôs+02].

"A closed termset, cs_i, is a frequent termset that is the largest termset among the termsets that are subsets of cs_i and occur in the same set of documents. That is, given a set $D \subseteq D$ of documents and the set $S_D \subseteq S$ of termsets that occur in all documents from D and only in these, a closed termset cs_i satisfies the property that $\nexists s_j \in S_D \mid cs_j \subset s_j$." [Pôs+02].

These closed-termsets are used to describe each document and every termset is associated with a weighting pair. The first element in the pair identifies the importance of the termset within the document itself and the second the importance of the termest within the index. These weighting pairs are used for ranking, but in addition the ranking mechanism uses three other factors. First, of course, is the frequency of occurrence of a term in a document, second is the generality of terms, with common words that occur in many documents being down-weighted, and third is a normalization procedure which ensure that large documents containing a lot of different terms are not unduly ranked higher than other documents.

2.5 Graph based IR Models

Although the standard approach in IR is to describe documents as a collection of words in individual index documents, there are other ways of modelling the contents of a document. One is to represent the contained text as a graph in which the nodes are words, whole sentences or even whole documents, and the edges represents the relationships. These can be determined in different ways, depending on the use case using statistical [BHQ03], syntactical [FCC07], semantic [MMD02], orthographic [Cho+07] or linguistic relevance. The high degree of freedom in the structure of a graph makes it possible to describe the non-linear and non-hierarchical structural formalism of natural language in a mathematical way. This, in turn, provides an excellent basis for different kinds of analyses about topological, statistical and grammatical aspects of a language [BL12b]. To support these possibilities, the underlying hypothesis of the graph-based IR models is that in a coherent text fragment, related words tend to build a network of connections which approximately matches the model a human being constructs in the process of discourse and understanding [HH76].

This kind of IR model is not new, since IR approaches based on graphs were being explored as early as 1969 by Minsk in the field of semantic IR [Min69]. Many approaches based on his results were developed in subsequent years, like neural networks, ontologies or associative networks. For example, one of the first neural networks, the Hopfield net, was used to model information in a graph in 1988 [Hop88]. In this network, information was stored in a single layer in the form of inter-connected neurons (the nodes) and their weighted synapses (edges).

A big advantage of these “connectionist” networks is that they fit very well to the Vector Space Model and the probabilistic IR models [BL12b] and greatly assist the ranking process. For example, in the context of the World Wide Web, the PageRank ranking algorithm uses a mechanisms which stores the web pages as nodes and their relations as edges to determine which website is referenced by the largest number of other web pages [Pag+98]. A web page with a lot of incoming relations is ranked higher than the others. Moreover, the information about the other web sites referencing a web site can be used in a higher-order scoring algorithm which also take into account the scores assigned to the other linked web pages.

Thus, the score assigned to a web page $S(v_i)$ can be influenced by the score of every directly related and indirectly related web page $S(v_j)$, where $Out(v_j)$ is the number of web pages referencing the site and δ is a so called damping factor which decreases as the distance to the actual node increases.

$$\boxed{S(v_i) = (1-\delta) \; + \delta \textstyle\sum_{j \in V(v_i)} \frac{S(v_j)}{|Out(v_j)|} \; (0 \leq \; \delta \; \leq 1)}$$

The use of this ranking approach has increased significantly in recent years, especially in social networks and recommendation systems [Sch+08] because graph structure are an ideal way to identify patterns [WH91] [Sin+09] in such information stores.

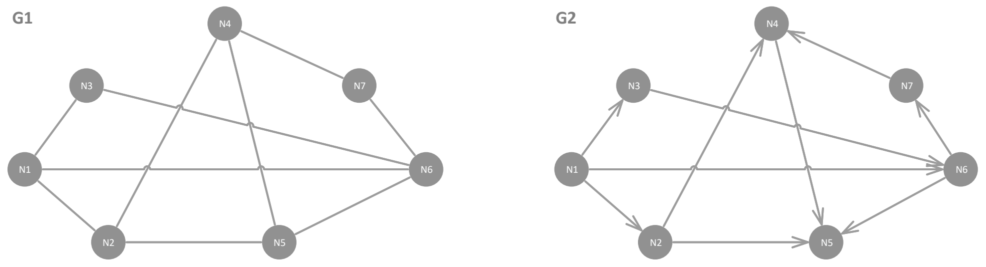


Figure 2.3: Undirected and directed graph [BL12b]

One of the weaknesses of the graph-based approach is its scalability, since performance can decrease dramatically as the size of the graph grows. Depending on the structure of the graph, calculating scores, or traversing the paths within the graph can be very costly. This happens, for example, if there are nodes in the graph which are referenced by a lot of other nodes and are then visited quite often in the analysis or searching process. In the case of graph-based representations of programs, this would be the case for primitive types, which if modelled as individual nodes would be referenced by nearly every class. However, if these structural characteristics are recognized and such bottle-neck nodes are avoided, the performance of graph-based IR approaches does not differ significantly from other approaches [BL12b].

3. Information Retrieval for Software Components

No man understands a deep book
until he has seen and lived at least
part of its contents.

– Ezra Pound –

Since source code is text, software components can be viewed as nothing more than sequences of strings just like books or web pages. Traditional information retrieval (IR) technologies can therefore be used to support the storage and retrieval of software components [SM83] [FB92]. However, although many of the techniques developed by IR researchers are helpful, such as correcting spelling error or identifying synonyms, traditional IR technology based on the Boolean Retrieval Model or the Vector Space Model leave a lot to be desired when used for code. For example, there is the mismatch between the properties of natural language and the technical structure and vocabulary of formal programming languages. This primarily creates the problem of how to formulate requests to such a system so that the users receives the relevant documents contained in the database and are not overwhelmed by irrelevant results because traditional free text approaches do not "understand" the special meaning of the concepts in source code. This chapter discusses the problems involved in using traditional information retrieval approaches for supporting the retrieval of software components and describes the range of possible solutions. It starts by enumerating the different retrieval approaches devised by researchers and then explains how these effect the problems of judging the relevance of software components. Finally, this chapter discusses a range of realization choices.

3.1 Software Retrieval Methods

Several promising approaches for searching for components using natural language have emerged over the years such as the profiling approach of Maarek 1991 [MBK91]. In his approach, Maarek extracted a certain number of indicators and combined them to create a profile of a component. This profile can be used as the basis for matching search queries to components to find relevant results. However, Maarek had to contend not only with the conceptual limitations of the IR methods available at that time, he also had to grapple with technical implementation details. In particular, full-text indexes as we know them today did not exist that time, so Maarek had to build his own index based on an uncontrolled vocabulary and clustering techniques. As pointed out by Mili 1998, software component search technologies were far from satisfactory in the past, and there was a lot of scope for further research [MMM98]. The problems with component retrieval technologies at that time were not just technical. Another well known problem addressed by numerous researchers was the so called "vocabulary problem" [Fur+87]:

> " No single word can be chosen to describe a programming concept
> in the best way ".

As a consequence, mapping natural language concepts used by software engineers when developing solutions to the technical constructs of programming languages is a challenging task since components can be developed in so many different ways [Mar+04]. Nevertheless, there are several approaches for mapping natural language to software components. For example, the natural language comments found in classes or methods can be used to infer mapping information, and some approaches like Exemplar [Gre+10b] leverage the accompanying documentation to support natural language searches for components, based on the text they contain. Unfortunately, however, the documentation accompanying components is often of poor quality and has its own semantic gap to the implementation [BMW94]. As well as searching for components by natural language, therefore, other researchers investigated approaches that build a special index based on their structural characteristics [ZW95]. This approach has a lot of potential because, as shown by graph-based IR models, software components have a lot of dependencies which could be captured in an index. However, this line of research has been neglected for a long time and has only recently been revived. Instead, the thrust of current research into software component search (or recommendation) has been to adapt and extend traditional IR methods.

In their seminal 1998 paper, Mili et al. subdivided the various retrieval methods into six different categories [MMM98].

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

However, Hummel et al. regard the sixth category *Topological methods* as ranking mechanisms [HJA07] rather than search methods, so we will not consider this category further here.

Descriptive methods

Descriptive methods are related to methods that describe components in a textual way. However, in contrast to traditional information retrieval methods which analyse the complete document (i.e. all the code) they only deal with abstract descriptions of components making references to a few keywords. The main objective of descriptive methods, therefore, is to classify components rather than to understand their semantics. When a search request is made, the keywords in the query are checked against a list of keywords that describes the component. The vocabulary problem causes a difficulty here, since the same concepts and classifications can be described in many different ways and different terms [Mit98]. As Mili et al. suggested, the best classifications are usually performed by human beings, such as software engineers or system administrator, but this is totally unscalable to the millions of components today's software search engines deal with.

Operational semantics methods

As the name suggest, these methods are related to the operational semantics of the executable code and use a unique feature of software called its "functionality". Podgurski and Pierce mentioned as early as 1993 that software components can be identified by only a few input- and output parameters. Using such characterizations of components it would, in their opinion, be theoretically possible to search for a component using only a few keywords and some input and output parameters. This would require the execution of components in the repository to determine whether they implemented the required input/output mappings [PP93]. However, there are several significant technical problems in doing this on a large number of potential third party components. One is the shear number of logistic issues involved in getting heterogeneous, third party components harvested from the Internet to compile and run in an automated way, given the many dependencies on specific operating systems, frameworks and other packages that need to be resolved before software can run. Another problem is related to security. In an open-source repository where software components are harvested by automatic crawlers from the Internet, no mechanism currently exists to identify in an automated way what functionality a component actually provides. Therefore, when executing components on a server as part of a functionality-based search, it is quite possible that malicious software could be executed. Nevertheless, solutions to some of these problems have been found, and execution-oriented search engines have a number of advantages as demonstrated by Hummel's test-driven search technology [HA04]. Innovations built into this technology included the use of traditional IR searches to pre-filter and rank components prior to testing and the use of sandbox mechanisms to protect the system against malicious code.

Denotational semantic methods

In contrast to the operational semantics methods which are based on executing components, denotational approaches aim to discern the semantics of components by analysis. These methods attempts to establish a match between the search query and software components using an additional document providing more information about the component than just the information in the source code. These methods are mainly related to the signature matching search algorithms.

Structural methods

The main difference between the structural methods and the other methods is how the software component is viewed. Instead of looking at the functional properties of the code, the focus is placed on the structural characteristics of the component such as the kinds of patterns used within it. As Mili mentioned, this is the most suitable approach when the goal is not to directly re-use components per se, but rather to take them as the basis for further development or as reference examples [MMM98]. In the case of direct re-use with copy & paste, it is more effective to match components to queries based on functionality rather than structure. However, as Mili points out, components with similar functionality often also have similar structure. Although this does not always apply in practice, there is certainly a strong correlation between the two [MMM98]. Research on this topic has primarily focused on the area of clone detection where approaches are being developed to infer the functionality of components from their structure to determine clones [QLS13].

Most code search engines developed to date fall into one or more of the six categories of component retrieval methods defined by Mili. With sometimes only small adaptations to standard IR technology, the first generation of code search engines are able to deliver quite reasonable results [FP94] [MBK91]. However, as Mili pointed out, they also leave a lot to be desired. They not only have difficulty coping with the syntax of programming languages, but also with their structural characteristics which have changed significantly over time. For example, the structural changes to programs introduced by object-oriented languages created major challenges for structural searches. With the advent of these languages it became possible to implement a function not only in a single class, but also to distribute the implementation over several classes. For an information retrieval system that is designed to compare a search request only with a single document rather than a collective of documents, this is a big problem.

Code search engines like Assieme [HFW07], Sourcerer [Baj+06], Portfolio [McM+11] or Codifier [Beg07] have tried to develop various "tricks" to support "structure-aware" searches that are able to take the distributed nature of modern software into account, but these have had only limited success. Other search engines have attempted to use the documentation accompanying components such as Exemplar or LISA, which analysed Twitter messages [AG15], to infer information about their content or quality.

Still other tools have explored the use of graph-based IR methods. For example GraPacc, a Graph-based, Pattern-oriented, Context-sensitive tool for Code Completion, is able give developers code suggestions directly in their IDEs based on the code they are currently working on and a graph-based representation of the already existing method calls [Ngu+12]. The creators of GraPacc claimed they were able to increase the precision of searches to 95% and the recall to 92%. However, this requires developers to have a basic knowledge of the framework being used and to have already written some of the implementation. Another approach in this area is ParseWeb, which supports queries of the form " source $\rightarrow$ destination " [TX07] and also takes sequences of method calls into account. In this approach, code examples are analysed for method calls based on their abstract syntax tree (AST). However, these two tools only consider the method calls and sequences within one class. The first approach to take the actual processes behind the method calls and the classes they involve into account was Portfolio. However, this required a fundamental change to the nature of search queries [McM+11] to take into account the fact that processes can have different paths. This is influenced, for example, by the different ways a class can be initialized or the parameters of methods.

Although this discussion has only covered the most well known search approaches, it already shows the diversity of approaches that researcher have explored and explains the strengths and weakness of the various approaches that have been developed to support component retrieval. It also reinforces Mili et al.'s contention that the quality of component retrieval approaches can be improved in basically one of two ways – by improving the structure of the databases used to store the components and by improving the way queries can be expressed [MMM98].

3.2 Search Queries

The way in which users have to formulate search queries has a big effect on the usability and effectiveness of code search engines. Especially the second aspect, user understanding, is a frequently underestimated factor because virtually all search engines today support the classical keyword based approach as a fall-back option if users do not specifically define another type of search. However, they usually also provide an “advanced” query language where it is possible to add more context about the keywords. For example, Koders offers the prefixes “ cdef:” and “ mdef:” which can be used to identify classes or methods with a specific name. If these prefixes are not used in a search query, Koders is unaware of the “meaning” of the following keywords and just searches for them “blindly” like any other keyword.

A search engine query language must provide the features needed to express the full context of the keywords appearing in a query. However, it must do so in a way that is as simple and intuitive as possible for users. Powerful search algorithms and database structures are not going to deliver significant benefits if the associated query language is extremely complex and difficult to use [CMS10]. The ease with which search queries can be formulated and understood has been shown to have a major impact on the perceived usability of code search engines [CMS10]. In general, however, the more information that can be conveyed in a query the more effective the corresponding searches are likely to be.

Haidoc developed a tool named Refoqus, which is able to predict the quality of the search results likely to be generated by a query and makes suggestions for reformulating it [Hai+13]. For example, when searching for code it is important to know what information relates to a class and what information relates to methods within it. Such a query reformulation approach can recommend how to change the query so that it will deliver the most relevant results. Another approach for improving query languages is the Program Query Language (PQL) developed by Martin et al.[MLL05]. The PQL allows users to formulate queries using structural information such as the order of method calls to an object. The combination of textual and structural information in search queries was introduced several years later by Wang, who developed a search query language in which it is possible to specify a combination of topics and dependencies [WLJ11].

In terms of simplicity and ease-of-use, it is not necessary for every user to understand the structure or the syntax of an advanced search query, but it should be possible for advanced users to formulate queries without too much effort, otherwise they will implement the desired functionality themselves in the same time they need to reformulate the query multiple times. Even if the search query language is quite simple to use, it might not always be possible for users to formulate the optimal query because they do not have the right information at search time. A common example of this is when users are searching for reference examples of a framework they have just started to use and know very little about the components it contains and how they are initialized, etc. This gap in knowledge is also known as Belkin's anomalous state of knowledge [BOB82].

In addition to the aforementioned problems, Croft identified two other major problems related to search queries [CMS10]:

- Search requests can represent very different information needs and therefore require different search techniques and ranking algorithms to find the best and most relevant results.
- A search query can be a very weak representation of the information needs of the user. This primarily arises when users are not able to formulate all their requirements in the query language, or more frequently, when users are too "lazy" to formulate long, detailed queries. Sometimes, the constraints

expressed in long, complex queries are so restrictive that there are no matching results.

If a search query does not return any results for a long, complex search request, users should be helped to formulate shorter search queries. Several approaches have tried to address this problem using mechanisms like stemming [Lov68] or spell checking [Kuk92] but these only have a limited effect in code search. Stemming, for example, does not consider code structure. However, other approaches for detecting semantic relations in text have been developed that can address this problem to a certain extent [Dee+90].

In summary, although a large number of approaches have been developed to improve retrieval methods, and many of the original problems have been solved, there still remains one big challenge to overcome – the mismatch between the topical relevance and the user relevance [Mil+99]. Recall, for example, the "CustomerManagementSystem" scenario from the previous chapter which returned a large number of results but only a few of them were relevant to the user. It is still an open research question to develop query languages to reduce this problem.

3.3 Relevance in Software Search Engines

In this section we consider again one of the most fundamental issues in search engines, the relevance of the returned results. Probably the single most important reason why code search engines are still not used by developers on a regular basis is the relatively low average relevance of their results. A study by Sim et al. showed that many developers primarily use Google to search for code [SCH98]. But as a general purpose search engine Google is completely unaware of the meaning of keywords within the textual body of a code unit. It treats code in the same way as any other kind of textual document and never checks in what context keywords appear.

However, in the field of code search engines this information is critical because it determines whether a keyword represents the name of a class, a method parameter, a variables used inside a method, or one of the myriad of possible constructs that can be designated in a program. When tested with the search example described in the first chapter Google returned one relevant component in the top five results. This is because user relevance is not considered because it does not take into account what the keywords in the search query mean. For example, Google makes no distinction between whether the *getName* method is really a method or just a string appearing somewhere in the source code, including the comments. Fortunately, in this case the search query delivered a decent result, but due to this hit-and-miss nature of result relevance, developers always have to have a careful look at the returned source code. Moreover, it does not obviate the need for the developer to search through each result to see if there is a *Customer* class which contains a *getName* method. In contrast to normal search engines it is much easier to specify relevance in this context. To judge user relevance, it is sufficient to indicate what the signatures of methods should look like and to which other components a particular component should interact with. In the field of code search, the geographical and temporal factors that play such an important role in normal searches only have a secondary role.

The classic measures of search effectiveness in traditional IR approaches are the precision and recall of the search results. The precision expresses what proportion of the returned results are relevant, while the recall expresses what proportion of all available relevant results in the index are returned in the result set [Cle61]. However, whereas in a normal search the user is often interested in obtaining multiple result to obtain multiple sources of information about a particular topic, in software development once a developer has one search result that is suitable (i.e. relevant) for reuse, he/she is rarely interested in having more results.

3.4 Realization Approaches

At the end of the last millennium relational databases were prevalent in almost every area of data storage, but for search engines they have some major weaknesses. This is why Doug Cutting's full text database, Lucene, was so widely and rapidly adopted, and today it is still the foundation for almost every text-based search related system [MHG10]. Lucene was primarily developed to efficiently store text documents and support high efficient searches over them. Even other types of databases, including relational and NoSQL database, frequently using Lucene in the background to increase their search performance for some specific fields.

Combining Lucene with other databases to overcome the former's weakness in storing relationships has also been tried in the field of code search. As mentioned above, Sourcerer was one of the first code search engines to combine Lucene with a relational database to store the relations between components [BOL14]. The core elements of the relational database schema they identified were projects, files, classes or entities, comments and their relations to each other. The metamodel defined by Bajracharya et al. can be seen in figure 3.1.

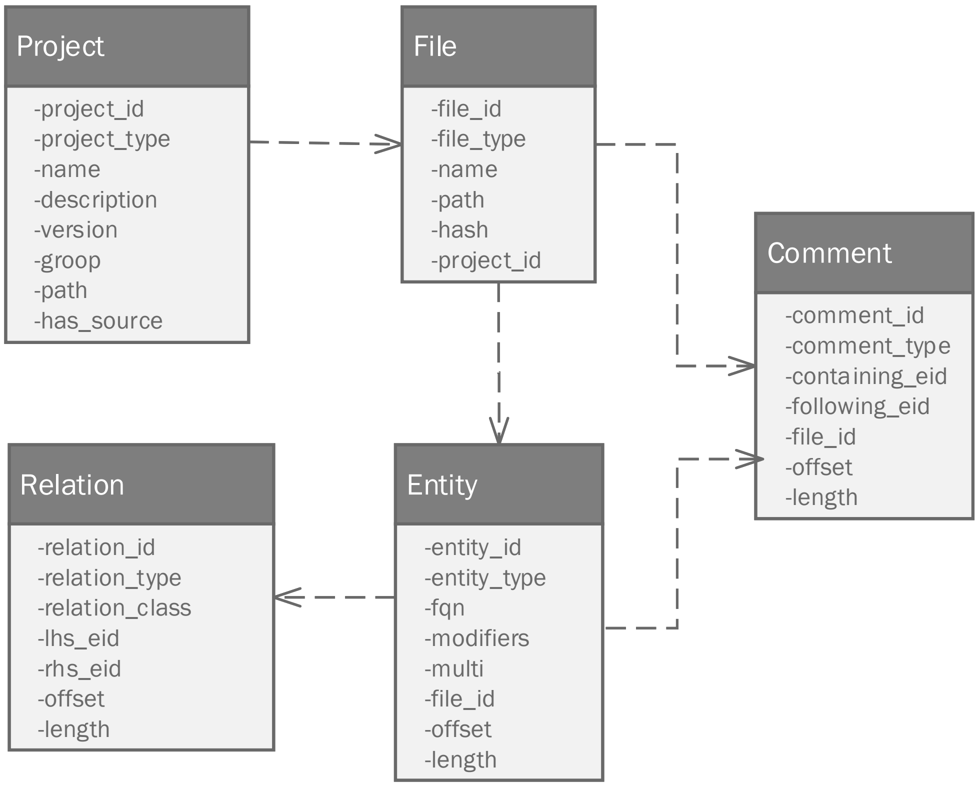


Figure 3.1: Sourcerer relational metamodel [BOL14]

A similar approach was explored by Hummel et al. with Merobase to complement the underlying Lucene index with a relational database [HAS13]. However, more attention was given to the relationships between classes, such as which classes are called by a given class, or in which methods a class is used as a parameter type etc. Hummel et al. also addressed the problem of missing related classes at crawling time. This is for example the case when classes are harvested from the Internet without the project context in which they execute. To cope with this issue the database schema included the ability to label classes as "candidates" to determine whether a class might be, but need not necessarily be, required by other classes. The final determination is then made after the end of the crawling and parsing process once a selection of candidates is available.

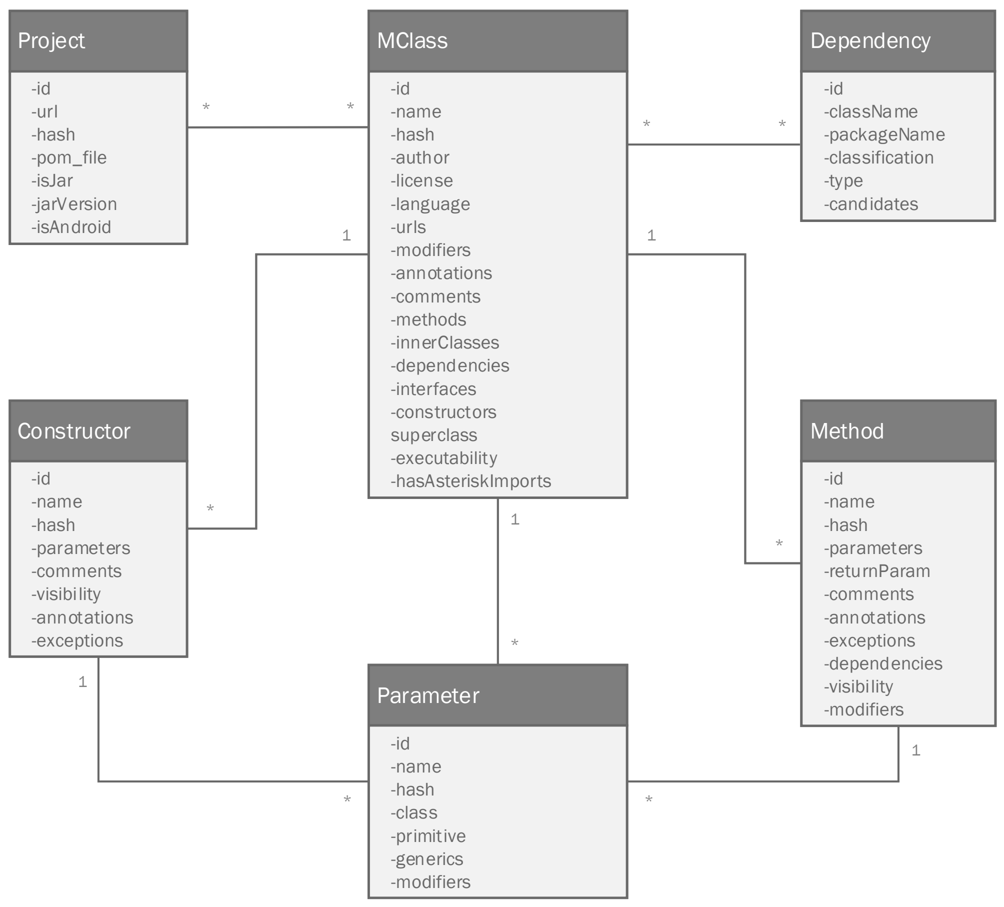


Figure 3.2: Merobase relational metamodel [HAS13]

Relational databases are only of limited value for storing source code modules and the relations between them. Although they can certainly store all the different kinds of relationships that appear in object-oriented source code (e.g. method parameter, superclass, global variable, etc.) they do not necessarily support efficient searches over this information. Depending on the query, certain kinds of searches typically involve a large number of joins to retrieve the desired results, and can lead to significant performance problems once the database exceeds a certain size. In the case of the Merobase, we observed a significant reduction in performance at a size of 1.5 Mio components. While this might not be a problem for a small, internally used search engine, for an online web search engine aiming to crawl and index all available source code from the entire Internet, this performance degradation is completely unacceptable.

It was precisely to address this kind of problem that graph databases, which are quite similar to the network model databases from the 1970's [Car85], arose in the late 2000s. "Graphs" have the advantage that they can not only be conceptualized in a simple way as a network of nodes and edges, they also have a concrete mathematical representation. Graph databases use the latter to organize the storage and retrieval of information. One of the most well known and widely used is Neo4J [Vuk+15], an open source (but commercially supported) graph database released in 2010. The information in simple graphs is stored exclusively in nodes and edges, but Neo4J uses attributed graphs which allow the nodes to store additional properties (i.e. attributes). This creates some additional options when deciding how to map real world objects to nodes. In addition, Neo4J allows one or more labels to be assigned to nodes to classify them. These labels significantly increase the efficiency of the search process since they allow pre-selections and/or scope restrictions to be performed. Like most other modern databases, however, Neo4J also uses an internal Lucene index to enhance performance. The role of Lucene in Neo4J is to map properties to nodes.

Since graph databases have a fundamentally different structure to relational databases, a new query language is needed to search in these. The query language supported by Neo4J is Cypher. This initially only had the ability to traverse over the graph from a starting node with the help of multidimensional indexes [MK14] but now supports a skip-list mechanism to speed up the traversal process. Original presented as part of SkipNet [Har+03], skip-list is a probabilistic data structure divided into several levels, where each level represents a ring to map a collection of sorted linked lists. Each node on the rings has a pointer to its predecessor and successor at every level. Although this structure can become quite large since each node in the ring needs $2log(n)$ pointers, it significantly increases the performance of certain kinds of searches. The actual number of ring nodes required depends on the created database schema and how many nodes needs to be created for each object. However, recent versions of Neo4J have reduced the node limit problem.

Since the emergence of efficient graph databases their uptake in software engineering has been rapid, but mainly in non-search related areas such as code inspection and evaluation tools. The first use of graphs in the area of code search was published by Horwitz in 1992 for analysing program-dependence graphs [HR92] in the context of local code management. Based on this concept, many other tools have since been developed to manage local code or to analyse process flows. The graph representation typically used to represent simple, locally-stored, object-oriented code is shown in figure 3.3. The corresponding data structures provide a powerful way for mapping cross-linked information and therefore for storing source code. In this simple example where *Class A* extends *Class B* and uses *Class C* by calling one of its methods, all relevant information about all involved classes is available for a search, since the search engine just has to go to the nodes in the neighbourhood. Suppose, for example, that a user wants to search for a *Class A* containing a specific method. A Lucene driven search engine could only provide results if the index contains a document with a field having exactly this combination of class name and method. A search engine driven by a graph database can also inspect *Class B* to see if this method is available and *Class A* can be added to the result set because it inherits the required method. Also, the graph-based search can usually outperform the text-based search, as Wang et al. observed [WLJ11].

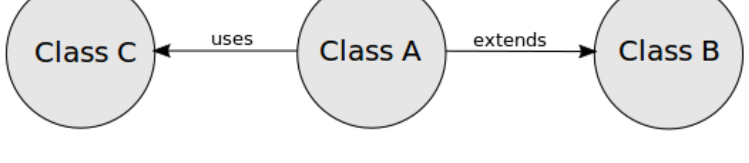


Figure 3.3: A simple example representing classes in a graph

This thesis is based on the premise that graph databases provide the best platform for implementing large scale, online code search engines populated by components harvested from the global Internet. The following chapters describes and evaluate an approach for doing this using Neo4J.

4. Code Search Engines

A world where everyone creates
content gets confusing pretty quickly
without a good search engine.

– Ethan Zuckerman –

This chapter describes the state-of-the-art in code search by discussing the key features of several leading code search engines. The first generation of dedicated code search engines were generally "web-based" since they were accessed over the Internet using some form of web interface. In recent years, however, the main focus has been on the development of so called "code recommendation" tools in the form of IDE plug-ins which make the functionality of "standard" code search engines directly accessible, in a context sensitive way, from developers' environments. As well as normal object-oriented software modules, some recommendation systems aim to provide such things as code snippets, frequently used classes and reusable test cases.

Many of the original code search engines and recommendations systems are no longer online or no longer the subject of active research. Perhaps the most well known, Google Codesearch, was shut down back in 2012 due to disappointing advertisement earnings. Others, such as Sourcerer is only available as a downloadable database and Koders closed down its freely accessible code search engine in June 2016. Nevertheless, these search engines are still discussed in this chapter due to the importance of their contribution. The chapter also discusses Portfolio [McM+11], Exemplar [Gre+10b], SymbolHound and Strathcona. While these may not be widely known or used they all contributed some interesting innovations to the field of software component retrieval.

All these code search engines attempt to return relevant software components or code snippets in response to a query. However, the developer must still overcome a certain cognitive dissonance in order to understand the results [Kru92], since the returned code is inevitably authored by different developers with different practices and programming styles. In addition, one major weaknesses shared by all these first-generation search engines is their limited consideration of classes related to, or in the neighbourhood of, the prime target of a search. Especially in the context of object-oriented programming languages, however, such relationships can have a major bearing on the quality of the search results, both in terms of precision (i.e. the extent to which developers can find exactly what they want) and recall (i.e. the extent to which all relevant (constellations of) code fragments are retrieved as search results).

4.1 Merobase

Merobase is a search engine developed by the Software Engineering Group at the University of Mannheim initially as part of the PhD work of Oliver Hummel [Hum08]. Inspired by the signature matching approach of Zaremski [ZW95], Hummel developed a signature-based retrieval approach [Jan+13] which falls into the category of structural retrieval methods described in the previous chapter. The basic idea is to relate the individual keywords within a code module via semantic relations that can be used to drive searches. Like many other search engines developed at that time, Merobase was built using the Nutch crawling tool and the Lucene full text search (FTSF) framework [MHG10].

The index supporting Hummel's signature-based retrieval method was populated by analysing code modules (in Merobase's case, programming language classes) on an individual, case-by-case basis. In other words, very little information about the relationships between classes was taken into account. Lucene is a highly specialized tool for indexing textual documents that is till today unequalled in its ability to perform keyword-based searches over text-based documents. However, since this requires documents to be indexed individually, unlike relational databases, Lucene indexes did not provided inherent support for associating documents to each other at that time. The only way in which any kind of relationship information can be supported is by designing "clever" fields in which related items of information from different place are concatenated. In a sense, Hummel's signature-based retrieval approach is based on capturing a few, carefully selected kinds of relationships within special fields so that Lucene can "recognize" them when searching for components. However, this approach is not only very inefficient and unscalable, it also fixes the nature of the relationships that can be considered in a search at index creation time. The special Lucene fields used in Merobase are shown in Table 4.1.

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 4.1: Most important fields of the Merobase index [Hum08]

These fields were used in the original Merobase index to represent individual components. Merobase also exploited the multi-field feature of Lucene which allows a field with the same name to occur multiple times. For example, multi-fields are used to store the methods of a class, since a class normally contains more than one method. Multi-fields allow each method signature to be stored in a separate instance of the field *methodSignature* so that signature-based searches for any of the methods can be supported. Also, to overcome the fact that parameters of a method can be arranged in arbitrary orders in search requests, in the *methodSignature* fields of the index they are stored in alphabetical order.

Once fields are present in the index structure they can be searched over using the full power of Lucene's comprehensive query syntax [MHG10]. However, because the fields were stored in a non-tokenized way in the first version of Merobase (i.e. they were not split up into subtokens that could be independently searched over) it was not possible to search for pure method signatures without method names or parameter names (i.e. by just taking the parameter types and returned vale type into account). This is because in a Lucene search, every keyword in the query must fully match one of the fields in the index. Therefore, over the years more fields were introduced into the index structure alongside the *methodSignature* field, such as the field *methodSignatureParamsOrdered* without method names. Another trick used in Merobase was to add a counter to method fields to count multiple concurrences of the same signature. For example, the signature of the *updateCustomer(int id, Customer customer):Customer* method of the *CustomerManagementSystem* class from the previous example would be stored in the Lucene index as shown in Table 4.2

Field	Content
methodSignature	mn:updateCustomer_rt:customer_pt:customer
methodSignatureParams	rt:customer_pt:customer

Table 4.2: Lucene field for the *updateCustomer* method

where *mn* is used as an identifier prefix for the method name, *rt* for the return parameter and *pt* for every input parameter. If the class were extended by adding another method with the same signature (i.e. with the same parameter profile and return value types), by for example a method such as *removeCustomer(Customer customer):Customer*, a counter is added to their method signature fields as shown in Table 4.3. This make it possible to define queries that search for components containing more than one method with the same signature.

Field	Content
methodSignatureParams	1_rt:customer_pt:customer
methodSignatureParams	2_rt:customer_pt:customer

Table 4.3: Counter for same parameter signature

Nevertheless, despite such tricks, the fundamental weakness of Lucene-based indexes related to relationships remains. Basically, a new type of field has to be added for every kind of relationship, or combination of relationships, that can be referred to in search queries. This not only adds tremendous redundancy to the information stored, but means that users cannot define queries containing relationship combinations which were not foreseen at index-creation time. Moreover, since Lucene does not support wildcard-searches within non-tokenized fields, a component's methods have to completely match the keywords used in signature-based query to be selected in the result. Classes whose methods have a similar method name or parameter names are not considered.

4.1.1 Merobase Query Language

Alongside the data structure used to create a search engine's index, another aspect of a search engine is the query language offered to users to formulate searches. This is one of the factors determining the perceived usability of a search engine and the degree to which it is seen as providing a valuable service to users. As mentioned above, Merobase users have access to the full Lucene query language to formulate searches based on the fields stored in the index. However, in general it is desirable to allow users to formulate queries in a way that resembles the languages they use in their everyday tasks. For software engineering, these are the primarily programming languages. Therefore, Merobase offers several extensions to the basic Lucene query language that enable the method information within queries to resemble the way methods are normally described in programming and modelling languages. The exact concrete syntax supported in this enhanced query language - the so called Merobase Query Language (MQL) - is inspired by the UML syntax for method signatures in class diagrams, which in turn was inspired by the Ada programming language. Using MQL, a search for a *CustomerManagement* system with the three methods from the example would therefore have the form:

```
CustomerManagement(
        getCustomer(int):Customer;
        addCustomer(Customer):void;
        udpateCustomer(Customer):Customer;
)
```

This MQL syntax is much more intuitive and usable for software engineers, enabling them to express queries that naturally resemble the normal representation of the artefacts they are looking for (e.g. code fragments etc.). However, as Hummel pointed out [Hum08] MQL still has significant limitations. First, MQL signature-based searches are strictly limited to the syntactic information, extractable from component "interfaces", leading to linguistic problems in the naming of the parameters – a manifestation of the IR vocabulary problem described in the previous chapter [Fur+87]. Second, because MQL is still essentially limited to the non-tokenized fields of the underlying Lucene index, strategies for reducing the vocabulary problem such as termsets from set-based retrieval approaches [Pôs+02], synonyms/homonym recognition capabilities in WordNet [Mil+90] or latent semantic analysis (LSA) [Dee+90] are not applicable.

4.2 Portfolio

Portfolio, a code search engine developed by McMillan, was one of the first search engines to explicitly consider relationships between classes [McM+11]. As well as storing static relationships between methods, Portfolio also maps the whole dynamic process behind method calls, including the full nested structure of sub methods calls, into a call-graph. This call graph is always available to users, regardless of how they reached the result, since it provides a better, faster and more intuitive way to understand the code. With its call graph, Portfolio addresses something that is frequently neglected in other search engines, helping users validate candidates returned in a search. Browsing such a graph is much easier than analysing the source code.

Portfolio also supports natural language queries and well known ranking techniques like the PageRank, the Vector Space Model and the Spreading Activity Networks (SAN). The PageRank mechanism is used to determine the global “importance” of a function in the overall call graph. In other words, Portfolio establishes which method is called most often by other methods, and makes this the centre of the net. Since the PageRank value is independent of the search query, this value can be calculated beforehand, for example directly after crawling, and stored in the database. McMillan calculates the PageRank $PR(F_i)$ for a function F_i in the following way:

$$\boxed{PR(F_i) = \sum_{(F_j \in B_{F_i})} \frac{PR(F_j)}{|F_j|}}$$

where BF_i is the number of functions calling F_i and $|F_j|$ is the number of functions which are called by F_i itself. The search process then is divided into several steps. First, the keywords of the search query are identified and used in a standard keyword-matching search, based on the Vector Space Model. The results of this search, most notably the nodes, are then annotated with several similarity scores calculated by the SAN. The lower a node’s SAN similarity score, the further away the node is from the actual function (the starting node). The SAN score for a node is calculated using the following function:

$$\boxed{SAN_j = \sum_i f(SAN_i \cdot w_i j)}$$

Here is the score for a node j, equivalent to the combined score of all nodes i pointing to the node j. $w_i j$, represents the strength of the relation between j and i expressed as a percentage (i.e to what extent the score of i affects j). Portfolio ranks all the functions of the call graph based on a combined similarity score:

$$\boxed{S = \lambda_1 PR(F) \; + \; \lambda_2 SAN(F) \cdot \lambda_i}$$

where λ_i is the interpolated value for all types of scores. Based on this value the top ten results will be returned. However, in Portfolio, the user is able to specify how many results should be returned.

4.3 Exemplar

Exemplar, an acronym created from EXEcutable exaMPLes ARchive, is also a search engine developed partly by McMillan [Gre+10b]. The main developer, however, was Mark Grechanik of Accenture Technology Labs. Unlike Portfolio, which addresses the reuse of source code, Exemplar focuses more on the reuse of executable and complete applications. For this purpose, Exemplar, combines three different sources of information about applications to deliver user relevant results. However, this assumes that the high-level requirements on an application match the semantic ones [HJD10], one of the main problems of denotational semantic methods. The three sources are textual descriptions (e.g. documentation), the API calls, that take place within each application, and the data flows between these API calls. The idea for Exemplar arose from the observation that the relationships appearing in search queries often resembles those found in the API calls and their implementations. This observation is closely related to the *software reflexion models* originally observed by Murphy et al. [MNS95]. Exemplar also deals with concept assignment problem in more detail – namely the problem that the semantics in the description of an application can deviate significantly from the semantics of the low-level implementation [BMW94]. Placing a keyword in the search query because it occurs somewhere in the description or

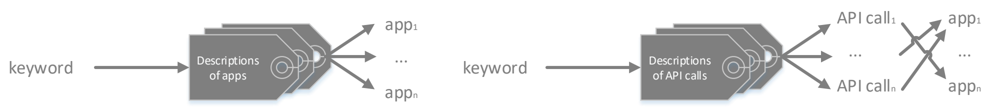


Figure 4.1: Query comparison between standard search engines and Exemplar [Gre+10b]

the source code, does not guarantee that the corresponding implementation matches (i.e. is relevant). To solve this problem, Exemplar combines the description with the corresponding abstract API calls. Although the idea to use API calls is not new [CJS09] [GCP07], it had never previously been applied to a large code base.

As in Portfolio, the search process is divided into several sequential steps. In the first step the keywords of the search query are matched to the documentation about the applications. In the next step, the results are analysed and the API calls $call_1 \ \ldots$ $call_k$ are determined from the documentation. These API calls are then compared to the called functions within the source code. If they match, the application is placed into the result set. The developers of Exemplar observed that different kinds of documentation about a component, such as JavaDoc or UML diagrams, which were typically written in a project by different individual persons, often use different terminology for the same concept. Their search engine is therefore able to address the vocabulary problem [Fur+87] by accumulating alternative terms and keywords in the index from the different kinds of documentation. To achieve this, every word occurring in the documentation $word_i$ is connected with an appropriate API call during the analyse phase $<< word_1, ..., word_n >, APIcall >$. Rather than store all this information in a single database, Exemplar uses several Lucene indexes in the background, one for the documentation and one for the API calls. Given all these features, the developer of Exemplar claims that the search engine is more efficient in finding relevant applications than Sourceforge [1] [Gre+10a].

[1]https://sourceforge.net

4.4 Sourcerer

Another academic code search engine first developed in 2006 and continuously improved since then is Sourcerer [Baj+06]. Unfortunately, however, direct code searches have not been supported online since in 2015, and only the source code and data set are currently downloadable. Nevertheless, Sourcerer has made a significant contribution to the evolution of code search engines. Like Merobase, the initial version was also built using Lucene indexes to store the individual source code files. However, subsequent version of Sourcerer were among the first search engines to enhance standard text IR techniques with source-specific heuristics. The first version of Sourcerer already supported various kinds of queries such as basic searches for components or functions, searches for code that uses certain components or functions and “fingerprints” corresponding to simple program structures or design patterns. These fingerprints are based on a vector-like representation of interesting attributes of entities in the source code, primarily to support structural searches. In the beginning, these fingerprints just contained several simple control structures in the source code, like concurrency, iterations and branching. Later, constructs were added related to the object-oriented aspects of Java such as classes, methods, constructors, etc., and micro patterns which capture simple design patterns used in the source code. After that, based on these fingerprints, comparison mechanisms from various IR methods [OH92] [B+99] were added to determine the similarity between two components [Lin+09].

In contrast to many other code search engines, especially the commercial ones, Sourcerer contains only Java source code, so that all the components underpinning the system are tailored to this programming language, like the crawler, the parser and the database structure. The infrastructure behind the search engine is presented in figure 4.2 and consists of five main parts: (1) a system to manage the crawler and the software repositories, (2) a system to parse the source code and to extract the features

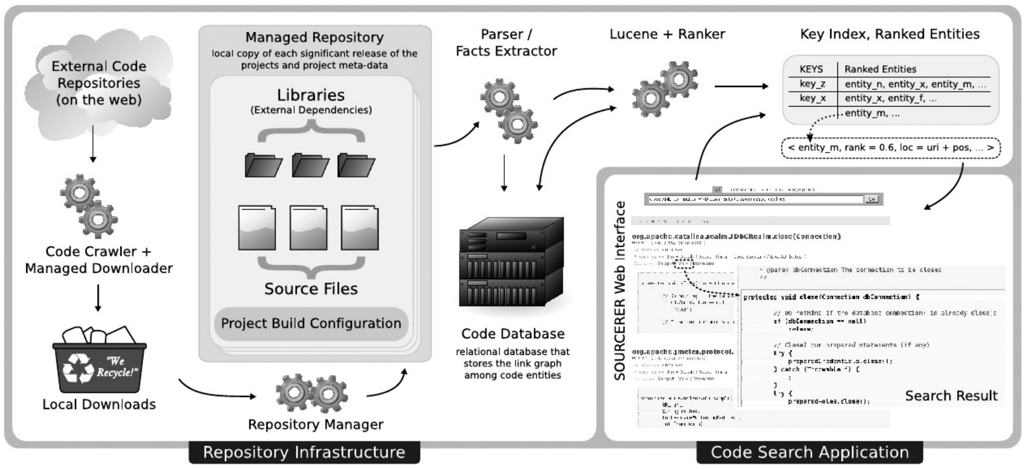

External Code Repositories (on the web)
Code Crawler + Managed Downloader
"We Recycle!"
Local Downloads
Managed Repository
local copy of each significant release of the projects and project meta-data
Libraries
(External Dependencies)
Source Files
Project Build Configuration
Repository Manager
Parser / Facts Extractor
Code Database
relational database that stores the link graph among code entities
Lucene + Ranker
Key Index, Ranked Entities
KEYS
Ranked Entities
key_z
entity_n, entity_x, entity_m, ...
key_x
entity_x, entity_f, ...
entity_m, ...
< entity_m, rank = 0.6, loc = uri + pos, ... >
SOURCERER Web Interface
Search Result
Repository Infrastructure
Code Search Application

Figure 4.2: Architecture of the Sourcerer infrastructure [Lin+09]

from the source code, (3) a relational database to store additional information, (4) various tools to search in the different databases and last but not least (5) a graphical web-frontend [Lin+09].

As shown in 4.2, Sourcerer was one of the first code search engines to use a relational database alongside a Lucene index to store code relevant artefacts. However, for performance reasons, only parts of the abstract syntax tree (AST) of the source code (several entities and their relationships) are stored in the relational database. The entities are unambiguous, identifiable elements from the source code, like classes, methods and constructors, while the relationships depend on what kinds of entities they connect (e.g. uses, extends, implements, calls, returns, override, receives etc.). To extract all this information, especially the different kinds of relationships, Sourcerer use the Eclipse Java Development tools (JDT) in the background. JDT supports automated dependency resolution, but only if a local copy of all the source code classes and the whole project is available. Even then, it is not always possible to resolve dependencies, for example, if pre-compiled jar files are missing in the project directory [BOL14].

4.5 Krugle

One of best known and most successful code search engines is Krugle [Kru13]. Unlike Portfolio or Sourcerer, Krugle is a commercial project which has evolved and stayed online for many years `http://opensearch.krugle.org`. The main company behind Krugle is Aragon Consulting Group, Inc, which sells the code search engine to third parties for internal use. One of the biggest advantages of Krugle is the large number of different programming languages supported, including widely used programming languages like Java, C++ or Ruby, and also markup languages like XML. In terms of search capabilities Krugle offers the general keyword based search mechanism as well as searches for such things like class definitions, function definitions, function calls, comments within the code or code snippets. The latter is an interesting and promising kind of search approach in which users insert a small piece of code into the search box and Krugle finds equivalent source code examples. Two basic levels of search functionality are supported for this kinds of search, exact searches where the search engine includes only exact matches in the result set, and fuzzy searches where less strict levels of similarity are acceptable. However, these two types of search are not mutually exclusive but represent grades on a scale, so users can select the level of fuzziness they require. This is analogous to Belkin's "state of knowledge" [BOB82] concept and allows developers to start the search process with only minimal information about the software they are looking for, and then to gradually increase the level of exactness as they learn from the initial search results. This kind of search was first introduced by Strathcona, but was not taken up by any other search engine except Krugle.

Since it is a commercial search engine, Krugle places particular emphasis on addressing enterprise needs. For example, it not only parses and analyses the source code, it also processes all the accompanying documentation like JavaDoc, Word documents, bug reports and commits in version control systems. This allows it to support many different kinds of use cases not just the “as-is” reuse of components or reference examples. It can also calculate many different kinds of metrics on the source code so that, for example, low quality components can be excluded from the result set or code clones can be detected. Code clones are a big problem especially in bigger companies and their automatic removal can significantly increase the productivity of developers and the quality of components.

Krugle’s index was built in much the same way as other search engines like Merobase using Nutch to crawl for components and Lucene to store the individual documents. The only difference in the process is related to parsing. Whereas Merobase parses software on-the-fly as it is harvested by the crawler, Krugle first stores the URL of all harvested components in a database, *crawlDB* [Kru13] and then performs the parsing process step by step. But before the parsing begins, however, the components in *crawlDB* are pre-ordered by special scoring values so that the most important files are parsed first.

The outcome of this parsing process is also somewhat unique. Instead of directly storing the AST information directly in the database it is first sorted in XML files which are analysed in a separate step to populate the Lucene index. The advantage of this intermediate XML format is that it is not necessary to write Lucene analysers for each different programming language. Instead, the Lucene index is created from the common, XML-based description of the components.

```
<krugleparse version="0.3">
   <uri>test/EndianUtils.udt</uri>
   <language>Java</language>
   <udt>
     <c b="0" e="803">
        <![CDATA[/* * Licensed to */]]>
     </c>
     <pkg n="org.apache.commons.io" b="813" e="833">
     <im n="java.io.EOFException" b="844" e="863"/>
     <im n="java.io.IOException" b="873" e="891"/>
     <im n="java.io.InputStream" b="901" e="919"/>
     <c b="952" e="1636">
         <![CDATA[/** * Utility code */]]>
     </c>
     <im n="java.io.OutputStream" b="929" e="948"/>
     <cd n="EndianUtils" b="1651" e="1661">
       <c b="1670" e="1748">
           <![CDATA[/* Instances should ...*/]]>
       </c>
       <fd n="swapShort" b="2038" e="2046"/>}
```

Listing 4.1: Krugle XML-Document [Kru13]

The main weakness of Krugle, like all the other search engines mentioned in this chapter, is that it only analyses components on an individual, case-by-case basis. It is also not possible to integrate elements of related classes into the search request. However, when classes are presented for analysis by the user as part of a search result set, the whole project is made available so that related classes can be easily inspected.

4.6 Koders - OpenHub

OpenHub, originally called Koders, is the second widely-known commercial code search engine alongside Krugle. However, unlike Krugle the core business of Open-Hub is not to sell the search engine to companies for internal use, rather the aim is to support open source projects such as the Mozilla Foundation. Unfortunately, OpenHub discontinued the openly accessible search engine in June 2016 in order to determine how to better support the open source software community. Although its search capabilities are relatively weak compared to other search engines, however, its index is one of the largest available. At the beginning of 2016 it contained 20.000.000.000+ searchable lines of code and was accessed by about 30.000 developers per day [BL12a]. Like Krugle, Koders also supports many different programming languages (in fact more than 40).

Koders shares the same basic weakness of all other search engines mentioned in this thesis that it only analyses components on an individual, case-by-case basis. Nevertheless, it is possible to search for various ingredients of a single component like method definitions, class definitions or fields such as global variables. Moreover, since the focus of OpenHub is the analysis of code projects, it supports searches for features related to project membership and properties not related to the execution of source code, such as the contributions of developers.

4.7 Symbol Hound

Symbol Hound was not primary intended be a code search engine, but it came popular for this purpose because of the kind of searches it supports and how symbols can be used in the search request. While most search engines keep certain symbols hidden from users when they formulate searches, like the wildcard character, in Symbol Hound they are all explicitly shown. This is especially important when formulating queries for languages like C++ or Ruby where the presence of certain special symbols can change the meaning of features. Examples are the pointer syntax and the "*" symbol in C/C++. While other search engines regard an asterisk at the beginning of the query, e.g. **customer*, as a wildcard character and thus return every result containing a word ending with *customer*, Symbol Hound takes this string literally and includes results containing a pointer to a customer variable. This feature is useful not only in relation to code files, but also for searching over postings in forums etc.

4.8 Strathcona

The previously discussed code search engines are all primarily known as web based. However, some tools are only intended for integration into IDEs. Strathcona is one of such tools for the Eclipse IDE [HWM05] which characterizes itself as a recommendation tool. Holmes et. al. observed that when implementing a system using a new framework, developers frequently face such questions as "*How do I initialize this*" or "*In what order do I have to call the methods*". The basic goal of Strathcona is to help developers find answers to such questions from within their IDEs by analysing the structure of the framework's code. To achieve this, Strathcona not only analyses the structure of the crawled code and the order in which methods are called to derive statistical information, it also analyses the new client code created by the developer to generate appropriate search queries. The starting point for query generation is always the current code under development in the IDE which could be either a class or a method. In the case of a class, the tool derives structural information such as the type of the class itself, the types of potential superclasses and the types of the global attributes. In the case of a method, the tool derives dynamic information as well as the messages that are sent and the objects which are instantiated. The names of types are completely ignored since they do not contain structural information. Therefore, if a developer is using a new framework and has already learned from the documentation of the framework that certain classes needed to be used first, as soon as they are opened in the IDE, Strathcona can start to gather the information needed to drive the search process to discover more information.

The results of searches can show the developer how the framework was used in other projects and if suitable for the developer's new project, can help include them automatically. At the very least, the developer receives information about how the framework is typically used in other projects. The main benefit, however, is that once the client code contains the first class initialization and method calls, Strathcona is able to provide very accurate results and help the developer to determine if any intermediate steps (e.g. method calls) have been missed out. This can significantly reduce the number of bugs that are introduced into a system even in cases where the developer is familiar with the framework being used.

5. Dependency-aware Metamodel

What we find changes who we become.

– Peter Morville –

A key component of any software engineering project is the creation of a model to describe the data types that the system in question has to manipulate. The model developed in this thesis constitutes a so called “metamodel” since it describes the language used to represent software modules in object-oriented programming languages. As well as fulfilling the basic criteria of supporting the core constructs of object-oriented programming (e.g. classes, interfaces, packages etc.) and the various dependency relationships between them, the metamodel was developed with extensibility in mind. In particular, one goal was to make it easy to add additional capabilities to the metamodel over time, while the other goal was to make it easy to support further programming languages in the future. Thus, although the focus of this thesis was on the Java programming language, the features of the metamodel were carefully designed to be generic to as many mainstream object-programming languages as possible. The next section describes the metamodel from a generic perspective, while the section following that describes how the metamodel can be used to represent Java constructs.

5.1 The Core Metamodel

The core concepts of mainstream object-oriented programming languages are very similar. They all basically revolve around the notions of *class* and *method* and the relationships that exist between them. However, as shown in figure 5.1, a "class" is not the most abstract concept in our dependency-aware metamodel. Instead, to provide more flexibility and extensibility, the most abstract concept in the model's inheritance hierarchy is *CodeObject*. This not only provides an abstraction that subsumes all source code files in a project, is also accommodates any other kind of artefact related to source code such as documentation. *CodeObject* therefore allows arbitrary new kinds of artefacts to be added to the metamodel in the future as soon as the corresponding parsing/analysis capabilities become available. In the same way that Java follows the principle that "everything is an object", therefore, we apply the guiding principle that "everything is a *CodeObject*".

There are three different kinds (i.e. subclasses) of *CodeObject* in the metamodel – *CodeProject*, *CodeComponent* and *CodeMethod*. The first subclass, *CodeProject*, essentially serves as a container of *CodeObjects* so that all the software-related artefacts found at a given root URL during the parsing process can be stored in one place regardless of whether they are related to one another or not. Of course, this includes complete projects hosted at one of the well-known software repositories. The ability to collect all of the code related elements found at the same address is an important feature of our approach since it allows the subsequent dependency resolution process to be much more comprehensive and flexible.

The motivation for distinguishing the two other kinds of *CodeObjects* lies in the observation made by researchers such as Gallardo-Valencia et al. [GS14], that even though methods are normally embedded within classes, software engineers are not always interested in reusing complete classes "as-is", but are often only searching

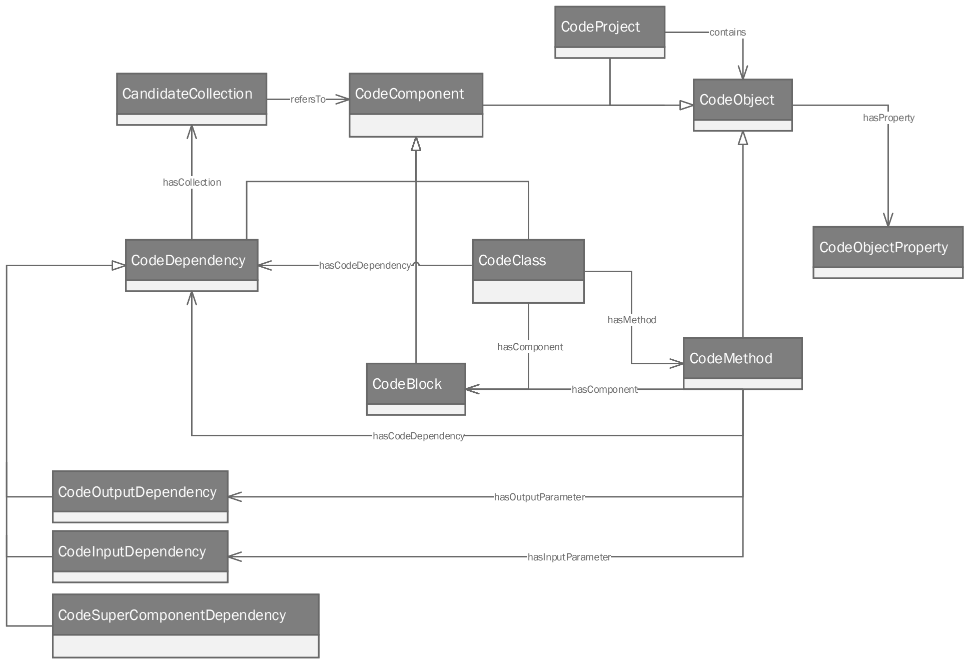


Figure 5.1: Core Metamodel

for only a single method to deliver a specific piece of functionality. Also, methods are not always reused “as-is”, but are more frequently used as reference examples. Another reason is the nature of the source code available on the internet. If only pure, complete projects such as those hosted at GitHub were analysed, the full context for a function would be available and could be directly assigned to a software component. However, on the internet a lot of context free, decoupled methods can be found, like methods which are presented and discussed in forums like StackOverflow[1].

Therefore, to include decoupled code that is harvested outside its original environment in our model, *CodeMethod* was added as one of the three forms of *CodeObject*. This allows methods and code snippets discussed in forum posts to be included in the repository. Since these code snippets are discussed and improved by many developers, sometimes over many years, they are often of high quality [SH13b]. It also allows the elements of languages that are not necessarily defined in the context of classes to be accommodated, such as JavaScript code fragments or functions

[1]https://stackoverflow.com

in functional programming languages. Of course, the fact that methods in object oriented programming languages normally belong to classes can be represented by means of the *hasMethod* relationship.

The third subclass of *CodeObject* is not *CodeClass* as might be expected but instead *CodeComponent*. *CodeClass* is in fact a subclass of *CodeComponent*. The purpose of *CodeComponent* is to allow further kinds of behaviour-encapsulating abstractions to be accommodated in the metamodel, especially components, since the basic goal of our approach is to support dependency-aware searches for software components. Probably the most well-known and accepted definition of "software component" is that of Szyperski and Clemens [Szy02].

"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.".

It would potentially have been possible to use *CodeProject* to represent components. However, since components can contain other components harvested from different root URLs, this is not convenient. It is clearer to retain *CodeProject*'s restriction to only contain artefacts from the same root URL and add *CodeComponent* to represent components that are not bound to this restriction. Among other things, this makes it possible to capture which classes belongs to which component or components (i.e. in the case when a class is used multiple times in different components).

However, it is not always the case that a class is an independently deployable unit, because it may have been harvested from an incomplete project or a discussion forum. Therefore, for the purpose of a code search engine the modern definition of software component proposed by P. Kruchten [Kru04] is more suitable:

"A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of a well-defined architecture. A component conforms to and provides the physical realization of a set of interfaces.".

This definition fits better to our approach because of the statement "nearly independent". This explicitly accommodates the case of reusing a class even if some dependencies are missing. The missing dependencies might either be irrelevant for the use-case in hand and removed, or the dependent parts might already be available in the user's system and fit to the reuse candidate.

The fact that a component can consist of many other components (e.g. in the style of the composite pattern) is one example of a situation which can give rise to cycles in instances of the metamodel. In the metamodel there are many other examples of relationships that can lead to cycles such as "*CodeComponent* → *CodeDependency* → *CandidateCollection* → *CodeComponent*". In general, the following types of relations in modern programming languages, can give rise to circular relationship patterns:

- Global variable types
- Method input parameters
- Method output parameters
- Locale method variables
- Super classes

The basic capability for creating circular relationship patterns comes from *CodeObject*. However, thanks to the different specializations of *CodeObject* it is possible to express whether the underlying source code is a software component or just a code fragment which does not fit to the definition of a component but can be regarded as *CodeMethod*. Since all code elements found on the Internet can have their own individual properties, the metamodel also supports two different ways of storing the properties – either directly within the element itself, or indirectly with an additional *CodeObjectProperty* element associated with the origin element. In general, a property that is shared by multiple elements is stored as *CodeObjectProperty*, whereas properties which are only relevant for a single element are stored directly at that element.

Another role of *CodeObjectProperty* is to allow processes to associate different elements with an object, for example, if a property is not known at parsing time, or if there are duplicates. However, as it is generally a bad idea to store redundant information in a storage system, our goal is also to detect these clones and avoid redundancy. Nonetheless, in terms of a globally-populated search system, even duplicates can provide information that is often worth keeping, like alternative URLs at which the component can be reached. This information is stored in its own *CodeObjectProperty* element, for example.

The role of the extends relation between *CodeDependency* and *CodeComponent* is to support the case when a component has a direct dependency to another component without a concrete visible dependency within the source code. This kind of dependency can be found in different plug-in mechanisms, for example when a Java class is initialized via the Java reflection capabilities. It is also useful for validation since, at source code analysis time, it allows relationships between individual code elements and their neighbours to be directly checked for validity. This makes it possible to immediately reject invalid code during the analysis process which helps to keep the index clearer and better structured. Since dependencies can occur in many different forms such as input- / output parameter, a super class, a local variable or a function call, the *CodeDependency* element is one of the most important elements in the metamodel. Although function calls are not a concrete dependency in the usual sense, they are conveniently captured by the *CodeDependency* element dependency since they are usually defined using local or global variables that point to the object through which the function can be called. However, there are also constructs where no variable is created before the function call, like the static method calls in Java. *CodeDependency* can also be used to represent dependencies supported by the Java reflection capabilities. For these and other reasons, *CodeDependency* is the element in the metamodel with the most subclasses.

Another important element in the metamodel is *CandidateCollection*. This addresses issues that became apparent when trying to augment a Lucene index with a relational database [HAS13]. Since it is possible to find several classes with the same name when crawling the Internet, or to find classes with different names but the same functionality, it is hard to determine the right dependencies directly if complete projects are not available. However, in order to support environment independent dependency resolution it is necessary to accommodate such classes and establish the relationships between them. *CandidateCollection* helps in this regard by proving a way of simply collecting all components or classes that fit to the first available information about the dependency. In a first step, all possible matches are added to the candidate list. This list is then analysed further in a second step, as explained in the next chapter, to reorder and restructure it. Until this is done the definition of a component may be violated since it is not guaranteed that the dependencies point to components. Adherence to the rules of components is only guaranteed once the second step has been performed. *CandidateCollection* and *CodeComponent* were added to the metamodel to support this temporary situation during the parsing process. This is also the reason why the element *CodeClass* is a subtype of *CodeComponent*.

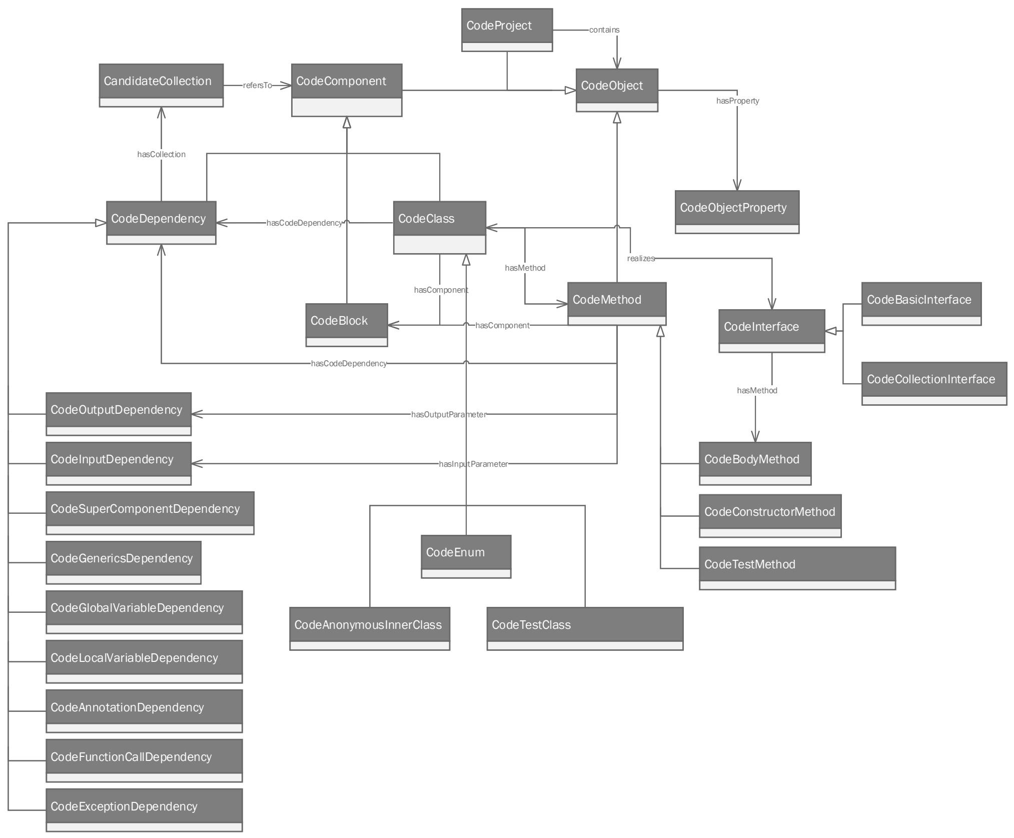


Figure 5.2: Java Specific Metamodel

5.1.1 Extended Metamodel for Java

The core modelling constructs described in the previous section can be specialized to represent the concrete features of many different programming languages. In this section we describe how we extended it to support the Java programming language since this language is the focus of the thesis. The Java specific metamodel is shown on figure 5.2.

Unlike many object-oriented programming languages, Java includes the concepts of interfaces to support multiple inheritance in a clean way. In the metamodel, interfaces are represented by the code element *CodeInterface* which, like *CodeClass*, extends *CodeComponent*. Interfaces cannot be seen as stand-alone components because they do not define executable code themselves, but abstract (or virtual) behaviour that has to be realized by at least one class in the system. The *realizes* association is included to identify the *CodeClasses* that implement them.

The metamodel distinguishes between two types of interfaces by defining two subclasses of *CodeInterface*, *CodeBasicInterface* and *CodeCollectionInterface*. *CodeBasicInterface* represents all the simple and normal interfaces, whereas the element *CodeCollectionInterface* accommodates all the instances of collection interfaces that are typed via generics – a peculiarity of Java. All standard interfaces of Java, like List, Map, Set, that are defined to be generic via a type parameter, are modelled using this element. It therefore simplifies the representation of generic interfaces and their realizing classes in the search engine.

The next Java-specific extension to the core metamodel is used to represent alternative ways of defining value classes such as "enums". In terms of the source code, enum definitions are just collections of keywords, but the Java compiler generates a normal class out of them with methods to access the values of the keywords, etc. An enum is therefore essentially a shorthand way of defining simple classes. This can be seen from the fact that it is possible to implement normal methods in enums or to instantiate enum keywords with parameters.

Another Java-specific form of class that has to be explicitly supported in the metamodel are "inner anonymous classes". These are classes which do not have their own name and are not defined in a separate file but instead are defined within the body of another class. They provide a convenient way of defining a local class using just a few lines of code when it is too laborious to define a new, fully-fledged class in the normal way. Such classes are most commonly used in the programming of user interfaces, where for example the system has to react asynchronously to events. In many cases, suitable listener classes (e.g. for reacting to mouse events) can be defined using just one line of code.

The final Java-specific specialization of *CodeClass* is the element *CodeTestClass*. This is used to represent classes which do not add any functionality to the system, but are used to test the functionality defined in other classes. In older version of JUnit, such as JUnit3, such test classes where fully decoupled from the classes they tested and needed to extend the class “TestCase”. Even if it is now possible in JUnit4 to integrate test methods directly in the class with the methods under test, it is still useful to capture classes containing tests to support test-focused searches [HA07]. Therefore, in the same way that *CodeTestClass* specializes *CodeClass*, the *CodeTestMethod* is defined as a specialization of *CodeMethod* in order to accommodate JUnit4 based testing methods in the search database.

The other subclasses of the *CodeMethod* are not actually Java-specific per se, as they can be found in many other object oriented programming languages, but they are of great importance in Java – the constructors and methods of a class. The concrete methods of a class are represented by the element *CodeBodyMethod* to distinguish them from the methods of interfaces which cannot be testing methods or constructors. Even though constructors are invoked, they can only be called once when a class is instantiated, unlike normal methods. Moreover, although they cannot have any output parameters, since they can have input parameters, they are still of interest in searches for methods.

Consider, for example, the system *CustomerManagement* where a *Payment* class is instantiated directly with a *Customer* object. Using a signature-based search it would not be possible to find this class if a developer is searching for a method like *makePayment(Customer, Double)* where the *Customer* parameter is the customer which has to be billed and the *Double* parameter is the value of the bill. However, with our approach, the customer is already in the *Payment* object due to the instantiation of the class, so the “parameter” of type *Customer* is available in every method contained in the *Payment* class. Therefore, for the purpose of supporting signature-based searches, if constructors were not handled as methods it would be much more difficult to search for this kind of structure.

Most of the extensions to the core metamodel appear as subclasses of *CodeDependency*, since this is where most of Java's idiosyncrasies occur. Examples are exceptions, which allow methods to react to unusual events, and annotations which allow additional information or instructions to be added to an element. For each of these specific kinds of dependencies additional elements are added to the metamodel as subclasses of *CodeDependency*.

5.2 Infrastructure of the Graph

The metamodels presented in the previous two sections represent the conceptual model we used also to specify our database schema. However, since graph databases allow elements to be connected in a different manner to relational database, the database schema does not have a one-to-one mapping of the metamodel. In a graph database it is possible to avoid some of the join tables or similar constructs that have to be used in relational databases. In this section we explain the steps taken to efficiently store information represented in the previously presented metamodels in graph-based databases such as Neo4J. We do this using the *CustomerManagement* example introduced in chapter Chapter 1. The basic search scenario in this example is for the method *addCustomer(Customer)*, which has a dependency via its input parameter to a *Customer* class, and is associated with this class via a *CandidateCollection* element. A representation of the metamodel elements needed to represent this example using the core metamodel is shown in 5.3

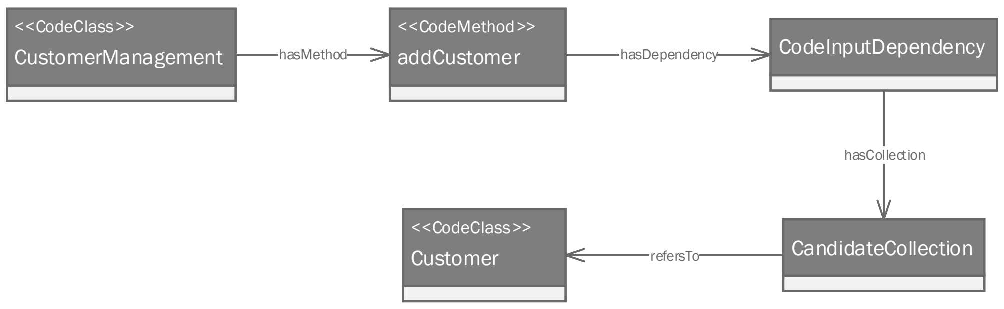


Figure 5.3: Example Application of the Core Metamodel

As already observed, the fact that there can be relationships between almost all elements in almost all directions means that instances of the core metamodel lend themselves to graph databases. In contrast to the pure metamodel, however, it is desirable for an entity stored in the database to store information not only about their immediate type, but also about all their supertypes. To do this we use Neo4J's feature of allowing nodes to have multiple labels to add all the supertypes to a node. For example, a testing method will not only have the labels *CodeTestingMethod* but also the label *CodeMethod*.

Of course, the name and supertypes of an entity do not provide enough information for a search engine to work effectively, so a lot more information is collected and stored as node properties or as an additional node labelled with *CodeObjectProperty*. The element *CodeObject* does not have any properties of its own, since it does not represents a concrete source code entity, but serves as the supertype of the other elements. All the other elements introduce different properties that are derived from the source code during the parsing and analysing process. In the following we will present the properties relevant to the search and dependency resolution processes.

Since *CodeProject* is not directly associated with code and merely serves to encapsulate other elements it has only two properties. One is the URL at which all the files it encapsulates (e.g. from a project) can be found and the other is the programming language in which most of the classes are written. It is always possible for a project to contain classes written in more than one programming language, so it is the language that the majority is written in that determines the language property for the project (i.e. the elements contained by a particular *CodeProject* element). For example, JavaScript is often included in Java source code via the JSNI (JavaScript Native Interface) mechanism, and a similar mechanism exists for including C++ code. To ensure that information about these programming languages is not lost, it is stored as additional property of the individual classes or interfaces. The programming language property of *CodeProject* is included to efficiently support searches such as "Which projects are implemented in Java?".

Most of the information is of course stored at the level of the individual *CodeClasses* themselves. In fact, each *CodeClass* element has the following six properties.

CodeClass	
name	the name of the class
lang	language in which the class is written in
hash	the hash-value of the class
url_n	the URLs of the class
fetchDate	the date the class was crawled and analysed
uniqueid	a unique id

Table 5.1: Properties of the individual class nodes in the graph

Some of these attributes, such as name, are read directly from the source code but others, such as the hash value, are calculated indirectly during the crawling process. Although the hash value property is not of direct relevance for users, it is useful for driving some of the analysis processes that take place in the background. For example, based on class hash values it is possible to detect simple forms of duplicates [Cho+02] of a class and avoid redundant information in the index.

Another generated value is the *uniqueid*. Even though the database generates its own implicit ID for every node, it is useful for each *CodeClass* to have a globally unique

ID as an explicit property in order to support environment independent dependency resolution. The *uniqueid* consists of the URL followed by the file name and the element name. Thus, the *uniqueid* of the *Customer* class from our example is –
`http://www.example.com/Customer.java/Customer`

To cope with classes that are nested within other classes, such as anonymous classes, in general the *uniqueid* stores the full containment hierarchy of each. This ensures that all classes have a globally unique ID. For example, consider the case of two variables with the same name but different types, where one is a method input parameter and the other one is a global variable. Without any additional information beyond the name the local variable is always used. However, if it is necessary in this method to access the global variable, in Java a "this." needs to be put in front of the variable name. Therefore, in Java first the local variables are checked and after that the global variables or the variables of the superclasses, etc. A similar mechanism is also used in our approach.

This hierarchical structure not only has a benefit for classes, but also for methods and other elements. For example, the *uniqueid* of a method has the form –
`http://www.example.com/Customer.java/Customer/addCustomer`

The only weakness of this approach is that only the first generated *uniqueid* is stored. In case of duplicates the *uniqueid* is not recalculated or regenerated. However, this can create problems if, for example, the first version of a class to be analysed is decoupled from a project (such as one retrieved from a forum post) and later a duplicate is found contained in a project. This make it impossible to use this unique id for dependency resolution.

The URL itself does not change, however, because Neo4J does not support multi-fields like Lucene which allow a keyword of a property to occur multiple times, the keyword or the URL "url" has to be extended with a counter $url_1 \ldots url_n$.

Some of the properties, like the url or the hash value are used for several elements. So they can be found again in the list of properties of *CodeMethod* as can be seen in table 5.2.

CodeMethod	
name	the name of the method
modifier$_n$	the modifiers of the method
hash	the hash-value of the method
url$_n$	the URLs of the method
containedIn	how many classes the method exists in
fetchDate	the date the method was crawled and analysed
uniqueid	a unique id
startPos	the start line in the source code
stopPos	the stop line in the source code
contentLength	the content length in characters of the method

Table 5.2: Properties of the CodeMethod

As with *CodeClass*, *hash* value is used to identify simple forms of duplicates. The next property, *containedIn* field, also used in duplicate determination, is just a count of how many classes this method can be found in. This is useful in statistical searches to identify widely used methods. Of course, to do this it is necessary to filter out all the getter- and setter-methods, as these methods frequently appear in multiple classes. The other properties mainly serve to help locate either the position of the methods in the source code or the classes to which the methods belong to.

There are no properties to store information about the parameters because this is stored in the graph via subclasses of *CodeDependency*. Input parameter dependencies are represented by instances of *CodeInputParameterDependency* and output parameter dependencies by instances of *CodeOutputParameterDependency*.

The other elements or nodes in the graph are mainly of the type of *CodeDependency*, which have two properties in common.

CodeDependency

classname	the name of the class of the type
fqn	the fully qualified name of the class

Since, for most kinds of searches, it is only necessary to know the kinds of dependencies that exists between two classes and all other properties can be obtained from the related class themselves, only the name and the fully qualified name of dependencies are stored to facilitate their identification. This significantly decreases graph navigation times when, for example, multiple dependencies of the same type exist, like the same input parameter type used in different methods.

Occasionally some sub-elements of *CodeDependency* have additional properties. For example, *CodeInputParameterDependency* also has a property to store the name of the parameter, the *CodeFunctionCallDependency* also has the name of the called function and *CodeGlobalVariableDependency* also stores the variable's position in the code so that they can be found more efficiently. Not every developer, unfortunately, places global variables at the top of a class. All other elements, like *CodeBlock* or *CandidateCollection*, have no properties as they are mainly auxiliary nodes.

This requirement could also have been addressed using various relationship types within the graph database. However, for several reasons we chose to use auxiliary nodes. The element *CodeBlock* was introduced because this kind of code structure can occur in different forms in the source code. Not only methods have code blocks, Java also supports so called static-blocks which are executed when classes are loaded and before they are instantiated. The elements of a static block do indeed occur within the context of a class, but due to the loading process they are decoupled from the rest of the class at run time. This allows us to determine whether, for example, a database has to be connected during the loading process or the initialization process.

To establish, which is required, a user just has to include the *CodeBlock* in the search request.

The second relationship that could have been captured solely using associations is *CandidateCollection*. However, if more associations were added to the node they would mix with the associations to the potential candidate. Not only that, as already mentioned we also associate *CodeObjectProperty* nodes to *Candidate-Collection* nodes to support the process of determining the right candidates. This has a distinct advantage, contrary to the approach with direct associations in our context-independent crawling approach. Of course, it would also be possible to use an approach similar to most other code search engines (Sourcerer, Portfolio, Exemplar) and always take complete projects from individual repositories like GitHub or Sourceforge. However, as already mentioned, Subramanian and Holmes observed that the code snippets found in forum post are generally of high quality [SH13b]. Moreover, the availability of complete projects does not always guarantee that every dependency is included because projects can be configured differently. So called automated build-management tools can complicate the task of resolving dependencies to third-party libraries. In today's build-management tools like Apache Maven, Gradle or Ivy, dependencies are defined in configuration files. However, this has the consequence that the individual library files are not stored in the project, but are stored centrally on servers or in local caches, and thus are also not pushed to the repository as in GitHub. Of course, it would be possible to write a parser for each individual build-management tool to get all the necessary information needed to resolve dependencies. An additional advantage of this approach would be that the right version numbers from the library would be available. However, given the rapid rate at which IT technologies change, it will not take long before the layout of build-management configuration files are updated. The corresponding parsers therefore need to be constantly adapted to keep up with these new layouts. For example, tools like Gradle are based on the version of Maven from 2004, but have their own configuration file with a different layout. The last issue to be dealt with are parallel referenced projects, which means projects in the immediate environment of the origin project. For example, in most of today's development environments like Eclipse, it is possible to put other local projects in the build path so that the classes can be directly used as dependencies. Of course, these projects are usually also pushed to the same repository and the project documentation explains that for execution the other project needs to be checked out, too. However, due to the vocabulary problem it is always difficult to get this kind of information. This is why DAISI employs an alternative approach to dependency resolution which provides some freedom in choosing the matching dependencies. Nevertheless, this approach also has some drawbacks. Since classes or code snippets are crawled and analysed independently from their project, the order in which they are loaded is unpredictable. Thus, in our example, it is possible that a *CustomerManagement* class is analysed before the *Customer* class. In this case, assuming no other *Customer* class exists in the index, it is not possible to identify a concrete dependency for the *CustomerManagement* class. Therefore, to handle this type of scenario we add a dummy *CodeObjectProperty* node to the graph containing the information available at analysis time. In general, this is the name of the class and sometimes the package name.

After that, when the *Customer* class, or any other class with the same name, is subsequently analysed it will be connected to the *CodeObjectProperty* node which is also referenced by the *CandidateCollection*. As illustrated in figure 5.4, over time every crawled class with the same class name and package name will be connected to this node. The figure shows the case where two *Customer* classes have been found. The same would happen in the opposite direction if the *Customer* class is the first to be analysed. In this case, the *CandidateCollection* node would be connected to the already existing *CodeObjectProperty* node. Of course, a lot of different classes may be available which have the same name but totally different functionality.

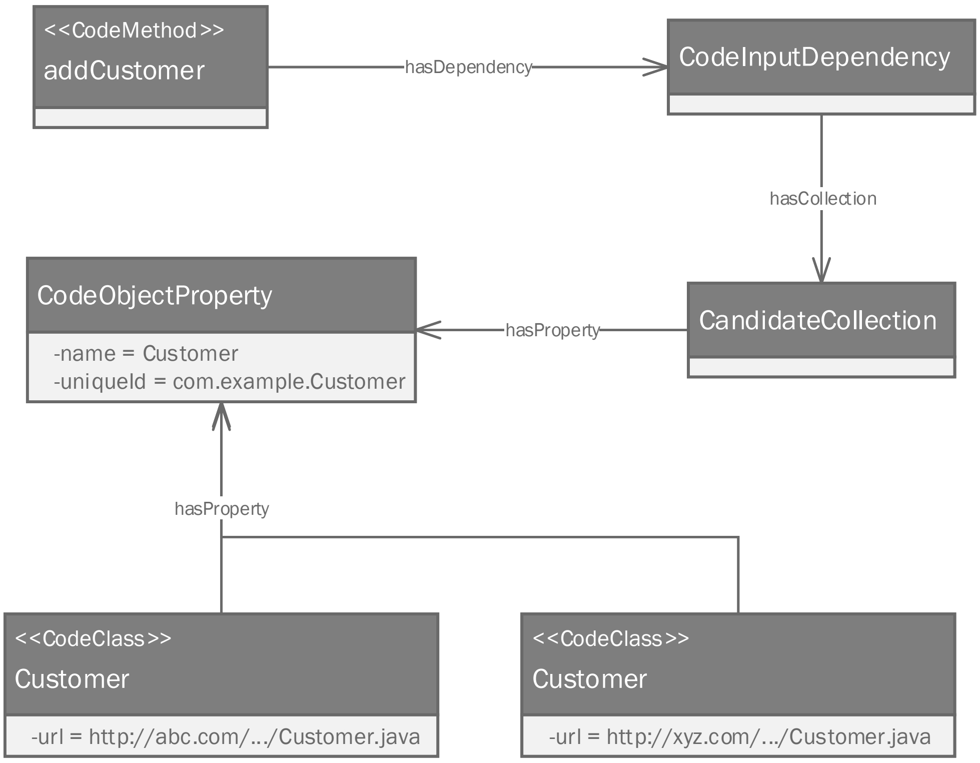


Figure 5.4: CandidateCollection example

5.3 Text Document Storage

Even though a graph is an ideal structure to store code related information, searches can still be computation intensive if large portions of the graph have to be traversed. However, since code is ultimately nothing more than text, storing code in a "Full Text Search Environment" (FTSE) framework has a lot of benefits as well. The most effective search performance is therefore obtained by combining graph-based and FTSE-based data structures. Many vendors of traditional database systems today augment their core data storage structure with FTSE (e.g. Lucene) indexes to enhance the search process. The usual approach is to prefix a search in the main data structure with a search in the FTSE index to narrow down the scope of data that has to be analysed [JZW09]. Modern graph-based databases such as Neo4J also use the same approach by preselecting candidate nodes from a Lucene index based on the keywords of the search request. To establish a direct mapping between the Lucene index and the nodes in the graph, every entry in the Lucene index contains also the node id of the graph. This makes it possible to read out the node ids of the individual search results and to take them as starting nodes to traverse the graph to refine the search results. However, Neo4J not only creates a separate Lucene index for each node type it does so fully automated. This means it is impossible to influence the underlying data structure or the information that is stored.

Since one of the goals was to retain the basic data search capabilities of Merobase, and in particular support MQL-based queries, DAISI also creates a so called “legacy index” in Neo4J. Although this is not completely integrated with the Neo4J data stores, and thus cannot be accessed using all the functionality of Neo4J, it can be given the structure and contents desired. The main disadvantage of not being completely independent of Neo4J is the loss of some query formulation possibilities in Cypher, the query language of Neo4J, but this can usually be overcome using query reformulations. Therefore, this disadvantage does not have a noticeable impact on our approach.

The big difference to the original Merobase architecture based on a single Lucene index is that in DAISI *several* Lucene indexes are created focusing on *different* areas. In parallel to the indexes automatically created by Neo4J, DAISI creates three different legacy indexes. The first index of *CodeClass* elements is very similar to the old Merobase index and incorporates some information about the code structure. The second index of identifiers discovered during the parsing process is mainly used to analyse the *CandidateCollection* elements to check whether a *CodeObjectProperty* node already exists. Finally, the third index of *CodeMethod* elements is mainly used to support searches for methods rather than classes or complete components. This index combines information in a different way to the first and thus delivers better results for searches that are not covered well by Merobase’s signature based search due to the choice of fields that are tokenized. As discussed previously, this problem can be addressed using Lucene's multifields feature in a separate index. Therefore, DAISI contains a separate additional index containing the fields of table 5.3 for the methods.

CodeMethod-Index

name	the name of the method
containedIn	name of the methods, where this method is contained
containedInCounter	in how many classes this method is contained
lang	programming language
inputParam	type of the input parameter
inputTypeFqn	fully qualified name of the input parameter (if available)
outputParam	type of the output parameter
outputTypeFqn	fully qualified name of the output parameter (if available)
url	URL of the class which contains the method
calledFunction	name of the called function
annotation	fully qualified name of the annotation class

Table 5.3: Index structure for *CodeMethods*

The field *containedIn* is defined as a multi-field to reference all classes where a method is contained in. However, DAISI does not just store the name of the class but also the referencing node id of the graph database, so that it can be used as the starting node of a graph traversal for a search. This node id is added to the name enclosed within two "#". For example, of the method *addCustomer* the value of the *containedIn* field would be `CustomerManagement#23#`. The first part is the name of the class and the second part is the node id (23). To tokenize this field we also added a Lucene specific `CharTokenizer` which splits the string at the hash character. This tokenizing makes it possible to directly read the id at search time. This provides a performance improvement, as it is generally expensive to read out additional fields from the Lucene index. Another option would be to always load whole documents, but this is not the case in general, as this would lead to a higher network and performance load.

DAISI uses the same format for all other fields for which it makes sense to store a direct reference to the node of a class or method in the graph. However, this is only done if we know which class is definitely the correct one. For example, the node id is only added to an *annotation* or *calledFunction* field if the right dependency is resolved. Until this is determined, DAISI only stores the name of the class or method without the id. The same mechanism is also used for the index of the *CodeClass* elements whose fields are presented in table 5.4.

CodeClass-Index	
name	the name of the method
namespace	the package name of the class
lang	programming language
protocol	the protocol how the class can be accessed
url	the URL of the file containing the class
methodSignature	the method signature of the contained methods, including method name
methodSignatureParams	the method signature only with the parameters
content	the source code of the class
comments	the collected comments contained in the class
interface	the name of the realized interfaces
interfacer	typified interfaces with generics

Table 5.4: Index structure for *CodeClasses*

This index of *CodeClass* elements is basically an extended version of the original Merobase index because it contains every field in the original index with appropriate extensions [Hum08]. For example, the original fields *methodSignature* and *methodSignatureParams* are extended by the node id to allow the corresponding node in the graph to be directly accessed. However, in the new version of the index we removed fields like *methodSignatureParamsOrdered* because there is an additional supporting index for methods and the parameters are now usually ordered. There are also two new fields – *interfaceSig* and *comments*. The *interfaceSig* field stores information about realized interfaces but in a way that supports Java's generics. This aids searches for collections which are only allowed to contain elements of a specific type. The *comments* field is used to separate natural language comments from the formal source code. In contrast to the original Merobase index structure, where all the text in source code files, including the comments, were stored in the *content*-field, DAISI separates the comments from the source code and stores them in a separate field. This makes it easier to include "Natural Language Processing" (NLP) retrieval techniques which are based on a semantic interpretation of the natural language. Of course, for the normal keyword based searches this decreases search performance slightly since two fields have to be considered. However, this decrease is not significant. All other fields are unchanged. Thus, the field *protocol* still describes how the class can be accessed, either via http, https, svn or git, and the field *content* still contains the bulk of the code for the keyword based search, but with a minor extension to reduce the amount of code that has to be stored.

6. Environment-Independent Harvesting

The ultimate search engine would basically understand everything in the world, and it would always give you the right thing. And we're a long, long ways from that.

– Larry Page - Founder of Google Inc. –

As two well-known founders of one of the world largest search engines wrote a few years ago, the demands on search engines are immense.

> "Creating a search engine which scales even to today's web presents many challenges. Fast crawling technology is needed to gather the web documents and keep them up to date. Storage space must be used efficiently to store indexes and, optionally, the documents themselves. The indexing system must process hundreds of gigabytes of data efficiently. Queries must be handled quickly, at a rate of hundreds to thousands per second." [BP12]

The demands on code search engines are similar, although they differ slightly in terms of the kind of objects they deal with, the crawlers, the indexing processes and the storage systems. Whereas fast response times are usually more important than result accuracy in general Internet search use cases, code search engines need to focus more on accuracy and ensuring that results satisfy user constraints. Therefore, the crawlers and deployed data structures are usually more complex in the latter.

6.1 Crawling and Parsing

Along with the metamodel and the database schema described in the last chapter, another key part of the DAISI search engine is the crawling and parsing process [GJ09]. The crawler is responsible for finding content that is suitable for the search engine, while the parser is responsible for analysing the content and incorporating it into the underlying database schema. Therefore, the crawler has to know which content is relevant for the underlying search engine and which content is irrelevant. The relevance of the content is determined by certain criteria or characteristics defined by the source of the information.

One way of identifying document relevance is through their type, since software documents of the same type generally contain the same kind of information or data. For example, every programming language has its own typical file extension. Another way to determine whether a document contains relevant information is by checking the MIME type of the document. MIME (Multipurpose Internet Mail Extensions) types were introduced with the internet [BF93] to describe the content of a document since file extensions are not always made available. The MIMI type therefore essentially delivers some meta data about a document. However, since file extensions or MIME types are not foolproof [Woo+96], the only reliable way to determine if the content of a document is relevant is through detailed analysis.

Nevertheless, the file extension or the MIME type provides a useful way of filtering out irrelevant documents and identify which document are worth analysing.

Identifying code in a forum post that is mixed with the normal text of discussions is more complicated. Fortunately, however, the tag "<code/>" is used in most forums and blogs where code is normally discussed. Therefore, DAISI's crawler identifies code based on this tag. The only way to identify the programming language used in such cases, given that no file MIME type is available, is to analyse and compare the code with the keywords of the different language. However, since DAISI focuses purely on the programming language Java, the crawler is configured to recognize only files with the extension ".java" and files with the MIME type "`text/java`". Additionally, to get the "<code/>" tags of websites, the MIME types "application/x-html+xml" and "text/html" are also considered. If a code fragment is detected on such a web site, an additional step is necessary to identify the programming language in which the code is written. Basically, the tokens in the code are compared to the keywords of Java and if they match the document will be parsed and analysed.

Instead of directly analysing discovered source code, however, we use the same approach as Krugle [Kru13] and first store the discovered URLs containing suitable code in a relational database. This database contains only three fields: one to store the URL of the discovered source code, one to store a Boolean value indicating whether this code has already been fetched by an analyser and one to store a Boolean variable indicating whether the analyser has successfully completed the analysis. The analyser gradually fetches URL's from this database for analysis. Since each source file is analysed independently of its project context the order in which the individual URLs are processed is irrelevant. Therefore, it is no problem to run multiple analysers in parallel since each analysis task can run completely independently. It is only necessary to coordinate the different threads when the nodes are stored to avoid redundant *CodeObjectProperty* nodes, for example.

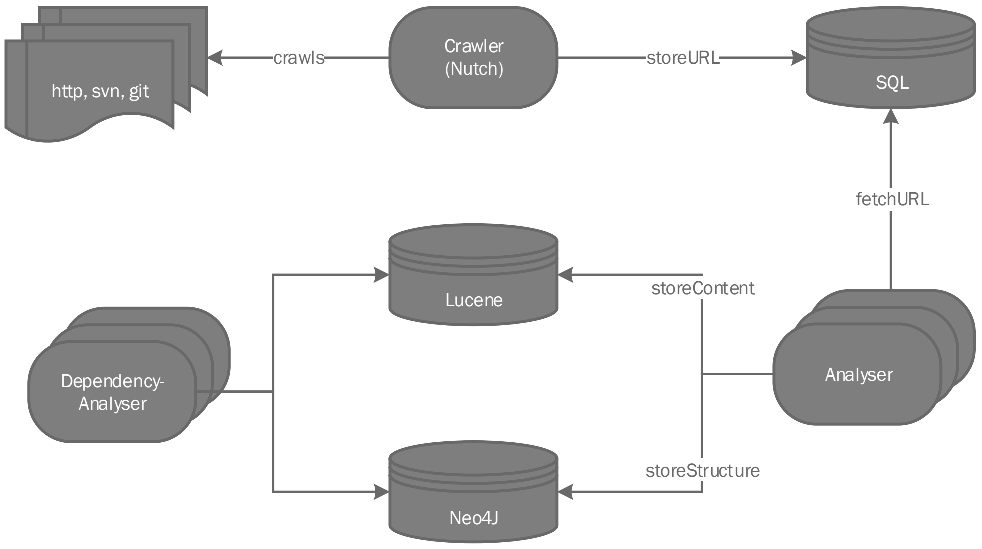


Figure 6.1: Crawling and parsing process of the DAISI search engine

6.1.1 Context-Independent Content Analysis

One of the reasons why it is harder to parse and analyse code components than text based web pages is their underlying context. Whereas web pages contain natural language text and are related to each other via world wide unique URLs, source code modules contain text expressed in formal languages and are not related to each other via globally unique links. Instead, the dependencies between code modules, which in Java are a combination of the package name and the class name, are only unique in the specific context of the software component. Changing the context can make the meaning of the dependency ambiguous. For example a dependency D might be unique in project A, but this project might also be used in project B where another class with the same name exists with a different implementation and functionality. In this case, if the context is changed the import statement of the dependency D is no longer unique. Java solves this issue using hierarchical dependency resolution. However, in terms of the global internet, where it is not always possible to find the whole context in which a software component was written, it is not always easy to determine the right hierarchy. The same issue arise if the dependency D does not exist

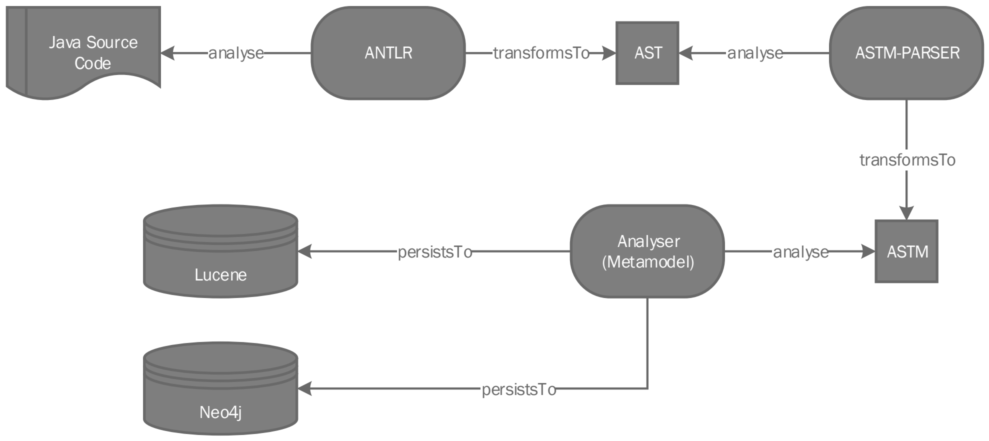


Figure 6.2: Process of the analyser and the different file formats

in the context it is used in. This can occur, for example, if the dependency is placed in a third-party library which cannot be found in the project context – something that can be observed today in all projects whose dependencies are configured via a dependency management tool such as Maven. However, DAISI's graph database approach provides mechanisms that make it possible to ignore the context of a software component. Various steps are needed to achieve context and language independence by transforming the underlying source of information to a uniform data format which can be analysed in a centralized, uniform manner.

Of course, the first step is always to download the code from its source. This could either be a location in the global internet or a SVN or Git repository in a company's internal network. To support these different kinds of protocols (http, git, svn, etc.) the necessary connector is determined from the fetched URL of the crawl database. In view of license restrictions and the academic characteristics of our project, we only collected source files from the global internet which are accessible via a normal HTTP connection. In the second step, the downloaded source code is then split into several parts and a language specific grammar is used to build an abstract syntax tree (AST) from the source code. An AST is a tree-based, abstract representation of source code in a specific programming language. At this stage a compiler is also invoked to determine the syntactic correctness of the source code based on the tree structure. This is a task for which the widely used ANTLR parser-generator system is well suited [Par13]. ANTLR was created in 1998 by Terence Par as part of his master thesis and has evolved over the years to the point where it is the most commonly used tool for parsing text in Java environments.

To use ANTLR to parse Java source code it is necessary to create an appropriate ANTLR definition of the Java grammar. Rather than store the information of the AST directly in the database it is first mapped to a platform and programming language independent format – the Abstract Syntax Tree Metamodel (ASTM), specified by the OMG [OMG11a]. ASTM is a generic metamodel which can represent the elements of all mainstream object-oriented programming languages. As soon as the ANTLR AST for a source file is created it is transformed into the standard ASTM format. Of course, this raises the question as to why DAISI does not use a document format directly corresponding to the metamodel to store the code in our central database. There are several reasons, but the main reason is that the metamodel was basically developed as a database schema, whereas ASTM was designed to represented all the elements of an AST in the XML Metadata Interchange (XMI) format. This format was designed by the OMG for the purpose of exchanging information between software tools and over a network. Therefore, XMI documents contain information, like header information, which does not need to be stored in a database for every element. Avoiding the use of ASTM as the underlying database schema also results in fewer nodes in the graph, because some elements no longer need to be stored in the database. In particular, nodes which are irrelevant for searches, such as statements in which values are assigned to a variable, can be omitted from the database. This obviates the need to traverse such nodes when a search is performed, thereby reducing the overall size of the database and increasing performance.

```
<Project>
  <files language="java"
      path="CustomerManagement.java">
    <import className="Map" packageName="java.util" />
    <fragments xsi:type="DeclarationAndDefinition:AggregateTypeDefinition">
      <aggregateType xsi:type="Types:ClassType" hash="-664476951|31736226">
        <opensScope>
         <declOrDefn xsi:type="DeclarationAndDefinition:EntryDefinition">
          <accessKind xsi:type="ASTMSyntax:Public" />
         </declOrDefn>
         <declOrDefn xsi:type="DeclarationAndDefinition:NamedTypeDefinition"
          typeName="CustomerManagement" />
         <declOrDefn xsi:type="DeclarationAndDefinition:NameSpaceDefinition"
          nameString="de.unima.informatik.swt.example" />
        </opensScope>
        <members>
        <member xsi:type="DeclarationAndDefinition:VariableDeclaration">
           <accessKind xsi:type="ASTMSyntax:Private" />
           <Identifier nameString="customers" />
           <declarationType isConst="true"
             xsi:type="Types:NamedTypeReference">
            <typeName nameString="Map" />
              <type
                  xsi:type="DeclarationAndDefinition:AggregateTypeDefinition">
              <aggregateType>
                <members>
                <member xsi:type="DeclarationAndDefinition:Declaration">
                  <declarationType isConst="true">
                  <type xsi:type="Types:String" />
                  </declarationType>
                </member>
                <member xsi:type="DeclarationAndDefinition:Declaration">
                  <declarationType isConst="true"
                    xsi:type="Types:NamedTypeReference">
                  <typeName nameString="Customer" />
                  </declarationType>
                </member>
```

```
                    </members>
                  </aggregateType>
                </type>
              </declarationType>
            </member>
                 <member xsi:type="DeclarationAndDefinition:FunctionDefinition">
            <accessKind xsi:type="ASTMSyntax:Public" />
            <Identifier nameString="getCustomer" hashValue="-1644953633"
              contentLength="67" />
            <returnType isConst="true" xsi:type="Types:NamedTypeReference">
              <typeName nameString="Customer" />
             </returnType>
            <formalParameters>
              <Identifier nameString="customer" />
              <declarationType isConst="true">
                <type xsi:type="Types:String" />
              </declarationType>
             </formalParameters>
            <body xsi:type="Statement:ExpressionStatement">
              <expression xsi:type="Expression:FunctionCallExpression"
                calledFunction="get" />
            </body>
             </member>
          </members>
        </aggregateType>
      </fragments>
    </files>
  </Project>
```

Listing 6.1: ASTM representation of the *CustomerManagement* class only with the *getCustomer*-Method

Even if some of the information is not persisted in the database it can be useful to support certain kinds of analysis. It is also possible to extend the underlying data structure to use other approaches like the call-graph searches supported by Portfolio [McM+11] or the multi-modal code searches defined by Wang [WLJ11], where it is necessary to know which statement or expression a method is called in. This is why in our process both metamodels are involved.

The ASTM evolved from the Knowledge Discovery Metamodel (KDM) [OMG11b] also defined by the OMG. However, the KDM is more suited to storing information from UML-like diagrams and is rather intended as an exchange format and intermediate representation for software systems and their operating environment as part of application life-cycle management. The KDM also defines metadata in the XMI format and is used mainly to represent entities, attributes and relations in an existing software system. However, when the OMG observed that transforming code to a KDM model structure caused the loss of a great deal of information, they decided to create a more code-related metamodel, the ASTM.

Like most OMG metamodels, the KDM and ASTM are based on the MOF infrastructure, also defined by the OMG [OMG11a].

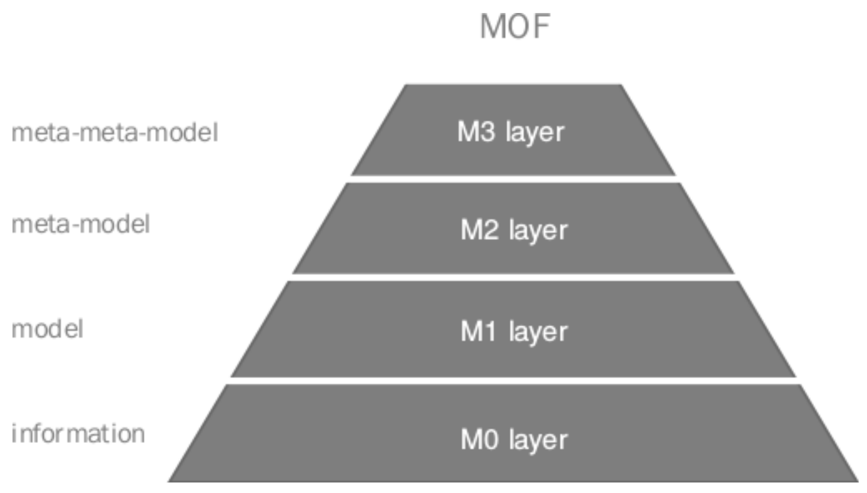


Figure 6.3: MOF pyramid defined by the OMG

The different layers capture different levels of abstraction in a system [Atk97] and both the KDM and ASTM metamodels are positioned in this MOF architecture in the following way:

Layer	Description	ADM Examples
Meta-metamodel M3	MOF (i.e., the set of constructs used to define metamodels	MOF Classes, MOF Attributes, MOF Associations, etc
Metamodel M2	Metamodels consisting of instances of MOF constructs	KDM UML profiles GASTM UML profile SASTM UML profile
Model M1	Models consisting of instances of AS model of COBOL language M2 metamodel constructs	KDM Data Model
Instances (examples) M0	Objects and data (i.e. instances of M1 model constructs)	AST model instances of source code of real application. KDM Data models instance of data base or data files

Table 6.1: MOF relationship of KDM and ASTM

The KDM is often used to transfer information from one tool to another or to transform a model into another. However, transforming source code to a KDM model leads to the loss of information because it focuses on the high-level semantic elements of a software system. For example, there is no way to model a for-loop in KDM. The ASTM was defined several years later to avoid this information loss and allow all the low-level implementation details like procedural logic or data definitions (exactly the information present in an AST) to be stored as well. Nevertheless, the ASTM has a strong dependency on the KDM to facilitate a simple mapping between the elements of both metamodels. This can be used for example to transform source code to UML diagrams and vice versa with minimal loss of information.

Likewise, according to the OMG, it should be possible to convert source code of one programming language into another language, a common problem in domains such as the banking industry with many legacy systems.

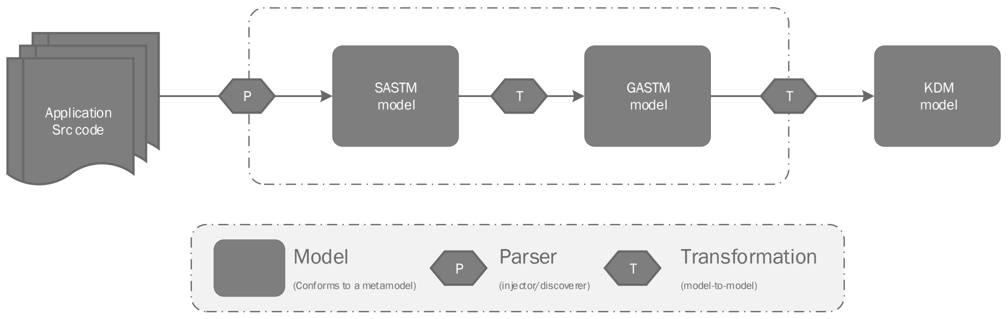


Figure 6.4: Process of transforming Source Code to a KDM to a UML Model

The ASTM is the only model needed in DAISI's parsing process, however, since it only has to capture information from the source code that is needed for the analysing process, even if it is not ultimately stored in the database. However, the use of the ASTM also creates future opportunities to include a subsystem to transform the information into a KDM related model for other analysis approaches or representation formats.

Since every programming language has it own characteristics and idiosyncrasies, the ASTM itself consist of two separate parts, the Generic Abstract Syntax Tree Meta-model (GASTM) and the Specialized Abstract Syntax Tree Meta-model (SASTM). The GASTM represents the core of the metamodel and contains generic elements that most programming languages have in common. The SASTM extends the core with elements specific to a particular programming language. Therefore, every programming language needs its own SASTM model, if it provides non-common programming structures and/or syntactical elements. To create the GASTM, the OMG analysed the most common programming languages and captured their common classes.

Domain	Data		Executable Code		Structure	Preprocessor
Programming Paradigm	Symbols	Types	Statements	Expressions		
Imperative Paradigm	Entry Definition Enumeral Definition Label Definition Procedure Definition Template Definition Type Definition Variable Definition Formal Parameter Definition	Collection Type Enumeration Literal Enumeration Type Exception Type Label Type Pointer Type Primitive Type Range Type Reference Type Structure Type Template Type Sequence Type Dimension Type Address Of	Block Statement Break Statement Case Statement Continue Statement Default Statement Expression Statement Try Statement Jump Statement Label Statement Loop Statement Return Statement Switch Statement Throw Statement Global Declaration	Array Reference Binary Expression Cast Expression Conditional Expression Enumeration Reference Identifier Reference Label Reference Literal Operator (Name) Pointer Expression Procedure Call Qualified Identifier Reference Range Expression Reference Expression	Compilation Unit Declaration Entry Point Procedure	Include Statement Include Unit Macro Call Macro Definition
Object Oriented	Class Definition Method Definition Member Definition	Class Type Inherits (possible relationship)				

Table 6.2: MOF relationship of KDM and ASTM

In addition to these various core elements, the OMG defined three different "scopes" in which semantic elements occur – Domain, Bindings and Location. The Domain scope defines all programming paradigms, the Binding scope defines ProgramScope, ProcedureScope, BlockScope and TypeScope as sub scopes, and the Location scope defines the two elements SourceLocation and SourceFile. For all these different categories, a corresponding root element exists, depending on whether an element belongs to the semantics, the syntax or the location (source).

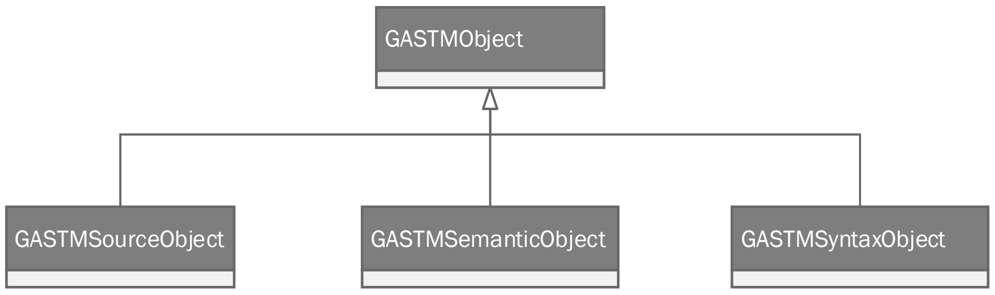


Figure 6.5: Root elements of the ASTM metamodel

Based on these scopes, for every necessary Java specific concept we defined a corresponding element in the SASTM model. However, since the GASTM already contains many elements relevant to Java, and also contains generic elements such as “annotation”, it is only necessary to add a few additional elements to support our search engine. Thus, the DAISI SASTM model only contains elements for import statements, static blocks, enums, interfaces and the different specific modifiers a method or attribute can have (i.e. native, volatile, static, synchronized, abstract and transient). In addition, we included two elements to better represent two Java specific types – exceptions as *ExceptionTypeReferenz* and primitive types as *PrimitiveTypeReferenz*. The latter is needed because in Java a wrapper class exists to each corresponding primitive type and these needs to be treated in a similar, but not identical, way. For example, depending on the Java version, in some case it was only possible for wrapper classes to determine equality using the equal method rather than the “==” operator. Also, with wrapper classes is is possible to check whether a variable is “null”, but this is not possible for primitive types. In DAISI, all kinds of primitive types are mapped to the same element of the ASTM model.

Element	Extends
Import	ASTMSemantics:GlobalScope
StaticBlock	DeclarationAndDefinition:Declaration
Static	DeclarationAndDefinition:AccessKind
Final	DeclarationAndDefinition:AccessKind
Volatile	DeclarationAndDefinition:AccessKind
Native	DeclarationAndDefinition:AccessKind
Synchronized	DeclarationAndDefinition:AccessKind
Transient	DeclarationAndDefinition:AccessKind
Abstract	DeclarationAndDefinition:AccessKind
Interface	Types:AggregateType
Enum	Types:AggregateType
ExceptionTypeReference	Types:TypeReference
PrimitiveTypeReference	Types:TypeReference

Table 6.3: Java specific SASTM Elements

Together, the model elements in the GASTM and SATSM metamodels provide all the information needed to map the ANTLR AST to an ASTM XMI file that can drive the code search engine. It is of course possible to extend these metamodels to accommodate additional features, or to create additional SASTMs to support new programming languages. The root element of the ASTM XMI file is always a "project" element, since the OMG defines this to be the top of the hierarchical structure regardless of whether the source file is located in a project or not.

```
<Project>
 <files language="java" path="http://example.com/Customer.java">
  <fragments xsi:type="DeclarationAndDefinition:AggregateTypeDefinition">
      <aggregateType xsi:type="Types:ClassType">
        <opensScope>
          <declOrDefn xsi:type="DeclarationAndDefinition:NamedTypeDefinition"
            typeName="Customer" />
          <declOrDefn xsi:type="DeclarationAndDefinition:NameSpaceDefinition"
            nameString="com.example" />
        </opensScope>
        <members>
          <member xsi:type="DeclarationAndDefinition:FunctionDefinition">
            <accessKind xsi:type="ASTMSyntax:Public" />
            <Identifier nameString="getName" hashValue="-245570169"
                contentLength="28" />
            <locationInfo startLine="9" stopLine="11" startPosition="11"
                stopPosition="4" />
            <returnType isConst="true"
                xsi:type="JavaSyntax:PrimitiveTypeReference">
              <type xsi:type="Types:String" />
            </returnType>
          </member>
        </members>
      </aggregateType>
  </fragments>
 </files>
</Project>
```

Listing 6.2: ASTM as XML

Every project can contain an unlimited number of *files* and every *file* an unlimited number of *aggregateType*s which represent the different types of classes, such as normal classes, enums or interfaces. The reason why a distinction between *files* and *aggregateType* is being made is that a file can contain more than one class, or class-like constructs, whereas an *aggregateType* cannot. The class name and the package name, if available, are placed under the element *openScopes*. Information about the visibility of the class is normally also placed under this element but is not shown in this example for space reasons. After the scope of the class, the *members* representing the individual syntactical elements of the source code are listed, like global variables, methods, etc. In the example above, however, only one of the methods is included – the *getName* method of the *Customer* class along with its return parameter of type *String*.

This XMI document can now be used to store the actual structure of the source code in a centralized way in the database. To include new programming languages only two additional tasks are necessary. The first is to build a new ASTM XMI document from the source code using, for example, an additional ANTLR grammar, and the second is to build a language specific SASTM model if language specific constructs need to be added to the index.

Since the ASTM model does not contain a validation mechanism to check if the source code represented via the model is syntactically correct, and the validation performed by ANTLR is optional, we also integrated a validation mechanism in the DAISI metamodel. This validation is performed via the aforementioned rudimentary neighbour check to identify incorrectly parsed code. For this purpose, every element type in the metamodel has a defined list of neighbour element types to which it can have relationships. So a method, for example, can have a *CodeInputParameterDependency* element as a neighbour but not a *CodeGlobalVariableDependency* element. The list of individual neighbour types allowed for each element type can be found in chapter 5. This list also defines the types of the allowed relationships in the graph. So a *CodeInputParameterDependency*, for example, can have a relationship of type *hasInputParameterDependency* to a method.

6.1.2 Handling the source code

Although the ASTM model, and thus the database, contains most of the syntactic and semantic information from the code, such as the method signatures or the relations between the different classes, it does not store every piece of text in the source code. Therefore, to support full text searches as well as graph based searches it is necessary to process the source in parallel and store it in a complementary Lucene index. This allows, for example, the Boolean retrieval model to be used to express queries with arbitrary combinations of keywords. The simplest way of achieving this would be to map the whole source file (i.e. every single string) to the *content* field used to store the code in the Lucene index. However, as Larry Page pointed out in his famous sentence "Storage space must be used efficiently..." any potential optimizations can have a dramatic impact on the size of the index and on the efficiency of searches. In fact, it turns out that certain strings in the source code can be neglected without having a visible reduction on performance. For example, when analysing the queries from the log-file of an internal search engine at SAP, Panchenko et al. observed, that most searches performed by developers were for method names (17%), followed by class names (13%), patterns (14%) and identifiers (9%) [PPZ11]. The term "pattern" in this context does not mean the structural relationships between classes, but rather queries in which developers used logical operators, like "<", ">", "=" or "==" to specify the relationships between the different terms in queries. In another study Bajracharya and Lopes examined the log files of Koders and discovered that 80% of the queries consisted of only one term [BL12a].

Therefore, one major optimization possibility is to avoid storing keywords that are rarely if ever used in search queries. A prime example are the method modifiers. When searching for methods, users invariably include the name of the method and sometimes also the parameters, but never the modifiers. Removing these modifiers from an index which potentially stores billions of source code files can save a lot of space. The same applies to the visibility modifiers of classes or global variables. When searching for variables it is usually irrelevant to users whether the variable can be accessed directly or only accessed via a getter method. In fact, most IDE's offer a service to automatically create getter and setter methods if they are needed. The same also applies to curly braces. The aforementioned studies showed that while users sometimes include round braces in queries when they are looking for methods with specific parameters, they never include curly brackets. Even if developers become familiar with using structural searches in the future, this would not happen in the context of pure text-based searches. On the contrary, other search mechanisms like interface based search would most probably be used which do not need curly brackets to be stored in the "content" field of the Lucene index. Based on these studies, therefore it is possible to ignore quite a large number of keywords and characters when storing the source code in the Lucene index.

Another issue affecting the relevance of the results in keyword-based searches are the comments embedded in the source code. The aforementioned analysis of Koders log files revealed that in many cases keywords in the search query only occurred in the comments. Although the corresponding result were topologically relevant in these cases, they were rarely if ever user relevant. Therefore, in the DAISI index comments are stored separately from the rest of the source code.

Other keywords that are not stored in the DAISI Lucene index are loops and conditions since these also rarely if ever appear in keyword-based queries. Such keywords are only included when users are searching for method calls, as it is possible in Portfolio. However Portfolio stores this information in a different database schema since it is not relevant for keyword based searches [McM+11]. Finally, we remove all white-space related characters from the source code, like tabulators, new-line characters or end of line markers such as the well-known ";" in Java.

Motivated by these aforementioned studies, the complete set of source code elements that are not included in the Lucene index is given in Table 6.4. On average this reduces the amount of information the DAISI database needs to store by 25%. This is a huge saving given the billions of source code documents in the internet that could potentially be harvested by code search engines.

Category	Terms
Modifier	public private, static, volatile,...
Brackets	curly bracket
Comments	every kind of comment, specified with // , /** or /*
Loops	for, while loops
Conditions	if, else
Space character	empty lines, white spaces, etc.
Delimiter	semicolon

Table 6.4: Removed terms of the source code for keyword based search

6.2 Graph-based Dependency Resolution

As described in chapter 5, the first step in the dependency resolution algorithm is to add a node to the graph for every potential dependency based on the available meta-information, such as class name or package name. However, in the first step no direct association is made between the element *CandidateCollection* and the potential dependency node. Instead, to determine the right dependency, the analysis process identifies all candidates via the nodes connected to the *CodeObjectProperty* node, the only node to which the *CandidateCollection* is connected in the first step as seen in figure 6.6.

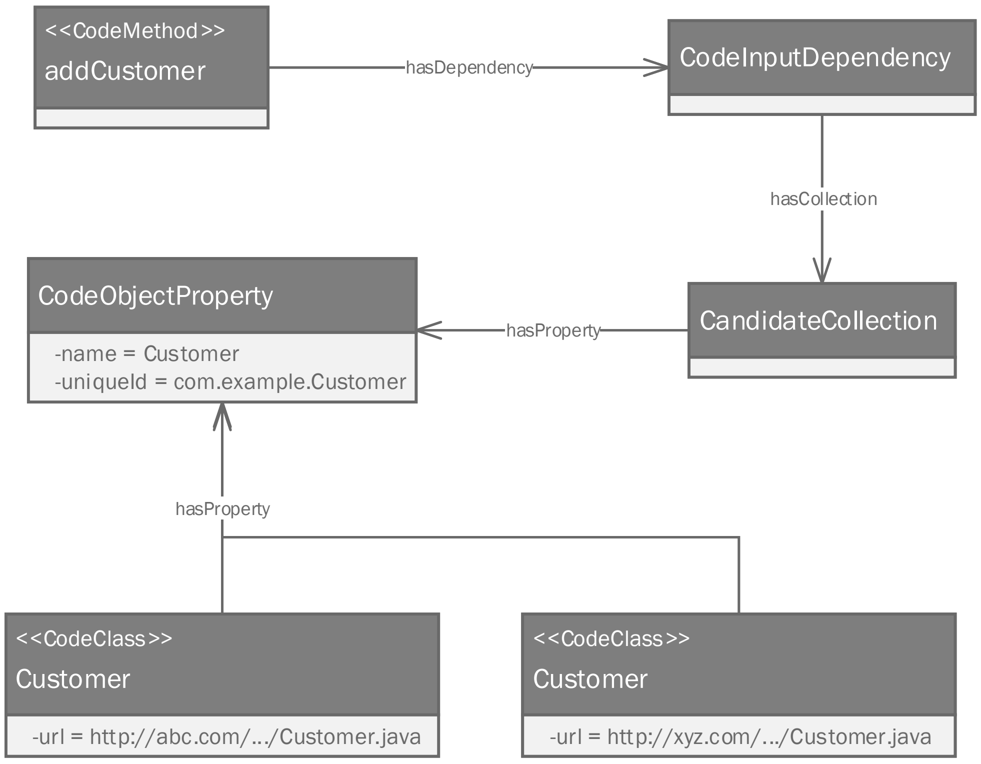


Figure 6.6: CandidateCollection example

In this picture, two different classes which have nothing in common except for their names are identified as potential candidates for a dependency. But since name selection is part of the creative design process conducted by developers, the names given to classes do not necessarily reflect their functionality, so it is not clear whether actually they represent a matching dependency. Therefore, it is necessary to perform an additional step to identify all appropriate candidates. This is achieved by gathering from the graph and the Lucene index all available information to determine which candidates match. The ones that do not match based on this information are omitted from the collection node of potential candidates. Examples of the type of information used in this steps are the called methods, information from the *FunctionCallDependency* node, access to global variables, the name of the author, the URL and the popularity (i.e. the number of incoming relationships from other *CandidateCollection* nodes).

The methods regarded as "called methods" are not just those from the currently analysed element, such like a method. Rather all methods called by the whole class are considered. Each method call is analysed to check whether a) the method is available to call and b) if the method signature matches the call parameters. If the method is not available or the method signature does not match the call, the class is omitted from the candidate list. The only exception is when a call to a *FunctionCallDependency* matches, but all other method calls from other parts of the origin class do not. In this case, we add the class to the candidate list because it is an appropriate candidate for the actual method call. If no candidates can be identified, it is necessary to use several dependent classes to obtain a complete component for the origin class, for example, by restructuring or adaptation. Even if this does not lead to the "as-is" reuse of the software it could still provide a good reference example.

The same process is performed for global variables, but with the exception that they are always added to the candidate list if they are available regardless of the visibility of the variables. As the method calls and the accessed variables are the only factors that influence whether a class can be compiled, they are also the only criteria used to determine whether the class is added to the candidate list. In fact, this only requires a restructuring of the graph, because even if a class does not make it into the candidate list, it will still retain its connection to the *CodeObjectProperty* node. Only an additional association of type *isCandidate* is created between the candidate collection and the class *CodeComponent* node, as seen in figure 6.7.

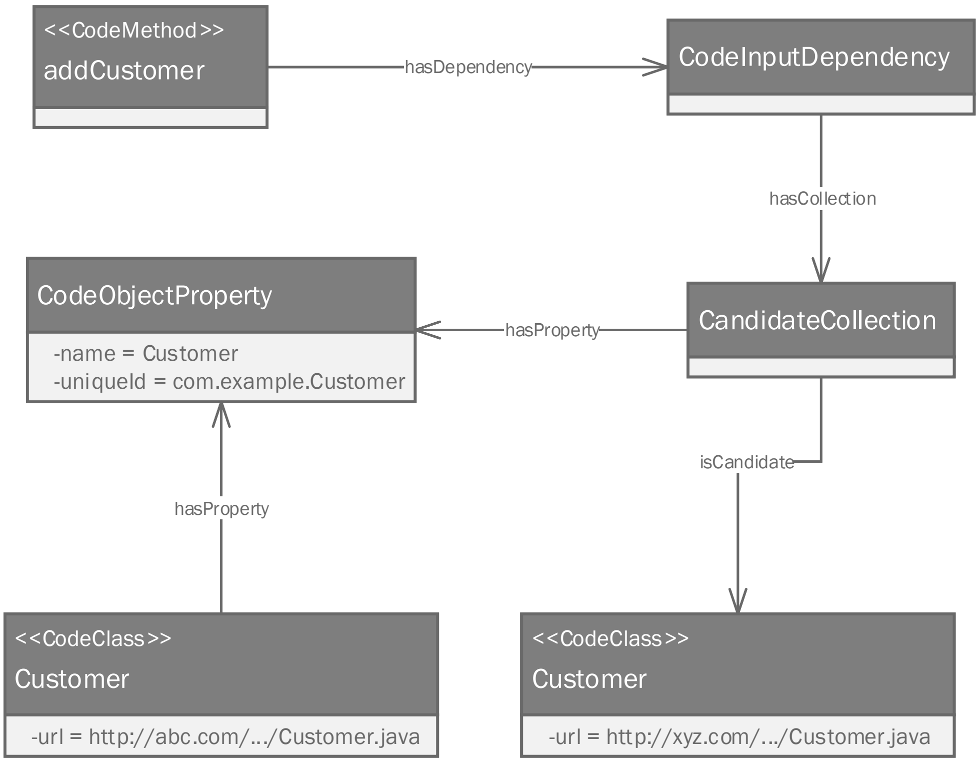


Figure 6.7: CandidateCollection example

Despite these steps it is still not guaranteed that potential dependencies will be correct. However, by reordering the classes connected to the node, containing the properties of the classes, direct "reuse" candidates will be selected first. All other forms of information, such as the name of the author, are used to determine the order of the remaining candidates. Nevertheless, since in most cases dependencies occur between classes within the same project, their root URLs in the database are likely to be the same. Failing that, they are likely to be located in a close project, so only the project names in their root URLs are likely to differ. This can be established from the generated uniqueid fields. Furthermore, in many cases a single developer wrote the classes within a component, so it is also likely that author name, if available, is the same. Finally, the popularity of a class is an indicator that it is likely to be a correct dependency. If a class is referenced by many *CandidateCollections* it is likely to be of high quality since it has been used and trusted by many other developers. However, a complicating factor is the version history of a class. One class with the same name, the same package, the same methods and the same author as another class might be a new version of that class containing bug fixes. This cannot be determined automatically from only the source code, however. Additional information about versioning is needed, which is sometimes available from the comments. Alternatively, humans needs to be integrated into the process to provide their judgment about the quality of a class. DAISI therefore contains simple community features which allow users to vote on the quality of a candidate via a like/dislike mechanism. The current version of this feature is very basic and can be improved in many ways. For example, it would be useful to not only be able to vote for a component, but also to influence further development steps by giving feedback about discovered bugs and providing hints about the structure of components etc.. Such capabilities could help establish whether components are worthy of reuse "as-is" in other systems

In contrast to the case where a list of potential candidates based on names already exists, sometimes there are no classes connected to the *CodeObjectProperty* node. In this case, the information about the called methods is again used to search for classes which have the required signature. Of course, there is a chance that this search returns results where only the method signature matches but not the functionality. Therefore, the user has to check the source code carefully to determine whether the required functionality is provided. The community functionality could also be used here to find matching dependencies. Moreover, this will improve the quality of the results because if a user declares that the first candidate dependency does not match, the second candidate can be carefully examined to see if it is a related class. In the long run it would be possible define a process which periodically identifies all components with poor scores and removes them. Alternatively it would be possible to resolve the dependencies using a test-driven approach, similar to the test-driven search of Hummel [HA04].

Regardless of how candidate components are determined, another advantage of this approach is that it is resilient to repository location changes. It is not uncommon for open-source projects to be moved from one location on the internet to a new one. A good example is the closure of the Google Code Projects site in 2016 in which all projects were moved mainly to GitHub. In this case, all resolved candidates and their URLs are no longer valid and accessible. Nevertheless, replacement classes are often available to ensure the compileability of the project. Of course, over time, the classes at the new location can be analysed and added again to the candidate list, but in the meantime other classes are often available to substitute for the temporarily missing classes. Also, this approach means that the system is less affected by zombie records. This was for example the reason for the ultimate failure and shut down of the well known UDDI repository which was built to provide a "yellow pages" system for web services [Atk+09].

7. Dependency-Aware Searches

There's nothing that cannot be found through some search engine or on the Internet somewhere.

– Eric Schmidt –

Although crawling and parsing are critical elements of a search engine, the part that end users directly experience is the query language. Query languages are therefore the gateway through which users access the functionality offered by search engines and thus have a major influence on their satisfaction with the service provided [CMS10]. However, today code search engines typically support only simple keyword-based queries, as revealed by Bajracharya and Lopes [BL12a] or Panchenko et al. [PPZ11], giving them little if any context information to identify components. Much more sophisticated forms of queries are needed to allow search engines to properly interpret keywords and find the optimal candidates for the problem in hand. This chapter focuses on the query language developed to support dependency-aware searches based on the graph-based code index described in the previous chapter.

7.1 DAQL

In programming languages, identifiers are used to designate a wide variety of artefacts ranging from packages and classes through methods and interfaces down to variables and constants. Therefore, without any further information a keyword in a query could match any identifier stored in the document schema used to represent components in the index. Most code search engines today therefore provide features for defining more context in the query languages. For example, in the Koders query language it is possible to indicate whether a keyword in a query should match to a class, a method or an interface using the the prefixes "cdef:" "mdef:" or "idef:" respectively. It is also possible to force the stemming of the keyword using the "*" character at the end of a keyword. However, the study by Bajracharya and Lopes [BL12a] showed these features are very rarely used (i.e. only 7% of the analysed search queries exploited these features)

The main reason why these prefixes are so rarely used at Koders is that 97% of the queries are issued by first time users who are unfamiliar with Koders query language. While it is essential to offer a rich set of prefixes for specifying how keywords should be interpreted, it is also important that the query language needs to be intuitive and simple to learn. For this reason, DAISI's query language for dependency-aware searches has been carefully designed as a conservative extension of the simple, yet powerful, Merobase Query Language (MQL) [Hum08]. The new, enhanced version, which we refer to as DAQL (Dependency Aware Query Language), is a conservative extension in the sense that all valid queries in MQL are also valid queries in DAQL [SA15]. In other words, DAQL subsumes MQL. This, not only makes the language accessible to users already familiar with the MQL, it also ensures that the set of queries that can be defined using DAISI is a proper superset of those that can be defined using the original Merobase technology.

Table 7.1 shows the set of prefixes supported in the DAQL query language.

Prefix	Kind of Search
method	search within the method index
comment	search for the keywords only within the comments
url	search for components of a specific URL
lang	search for components in a specific language
id	search by the id of a node
mql	old MQL search of the Merobase (added automatically)
defs	new kind of search

Table 7.1: Prefixes for the different search capabilities

The simplest prefixes are “url:” and “lang:” since these simply cut down the scope of a search to components written in a specific language or originating from a specific URL. The prefix “comment:” exploits the fact that DAISI separates the comments from the code so users can search for keywords that appear only in the comments. This provides a very simple form of natural language processing, but more sophisticated mechanisms could easily be added in the future. The prefixes “mql:” and “defs:” are used to explicitly indicate whether a query should be interpreted as an old MQL query or as a new DAQL query. If neither is added, “mql:” is taken as the default. So for example a query for a *CustomerManagement* system like

```
CustomerManagement(
        getCustomer(int):Customer;
        addCustomer(Customer):void;
        udpateCustomer(Customer):Customer;
)
```

will be identified as an MQL query and the search performed with the original Merobase semantics. The automatic detection of MQL queries takes place using regex pattern “.*(.*(.*).*).*”. Thus, at a minimum of two opening brackets and two closing brackets must occur in the query, as it only makes sense to search via the MQL if a minimum of one method is specified in the query. This search request is then mapped to the *methodSignature* and similar fields of the Lucene index mainly according to the original Merobase algorithms. Due to some minor changes we have made to the underlying index structure, we also had to slightly adapt the original algorithms.

To allow queries to leverage the relationship information in the graph database, DAQL divides them into two distinct parts, the first part dealing with class definitions and the second part dealing with relationships. Both parts have to be introduced by their own prefixes. Class definition are introduced by the prefix “defs:” in the intuitive, but extended, MQL style while relationships between classes are introduced by the prefix “deps:”.

In DAQL, the same query as above, but extended with the information that the *Customer Management* system needs a dependency to a *Customer* class, would have the form:

```
Defs:{
    C1:CustomerManagement(
             getCustomer(int):Customer;
             addCustomer(Customer):void;
             udpateCustomer(Customer):Customer;
    ); C2:Customer(
            getName():String;
    );
}
Deps:{
        C1→C2;
}
```

As this example illustrates, DAQL uses the same formatting style as MQL concerning class definitions except that every class definition is extended with an identifier, “C#:”, where the “#” represents a counter value. The “C#” identifier not only assigns the following MQL query fragment a unique number, it also identifies it as a class. It is also possible to identify the construct as an interface using the “I#” prefix. The unique identifier is used afterwards in the optional “deps:” segment of the query where the desired relationships between the classes and interfaces are specified. Several different types of dependencies are supported such as the “use”, the “extends” and the “realizes” relations. To represents the different types of relationships between classes, DAQL uses different types of arrows as seen in table 7.2. These were designed to roughly resemble the symbols for the corresponding relationships in the UML

Relationtype	Example
Association	C1 -> C2
Extension	C1 -|> C2
Realisation	C1 - -> C2
Method-Call	C1.addCustomer() -> C2.getName()

Table 7.2: Relation types within the query

Thus, in the query above, the *CustomerManagement* class should somehow "use" a *Customer* class somewhere in its body. In terms of method calls, it is currently only possible to specify which methods have to be called from another method as illustrated in the example. This specifies that the method *getName()* has to be called from within the method *addCustomer()*, to determine whether or not a customer is already in the data set. Therefore, currently it is not possible to specify that a method call has to occur somewhere in the code of a class.

Nevertheless, this simple set of relationships allows most search scenarios to be supported and quite complex queries to be formulated. In particular, the ability to formulate search queries which are able to return multiple classes, obviates the need for users to perform separate searches for every desired class and evaluate whether the separate results fit together and/or need adaptation. The only disadvantage of this kind of query is that they can became quite long and complex. To counteract this problem a mechanism for defining queries in a graphical form is presented in chapter 8.2.

7.2 Search Types

Now that the different data sources (i.e. Lucene indexes, graph database) and the DAQL query language has been defined in this section we present the main search types supported by DAISI.

Type 1: Keyword-based search

The simplest and yet still one of the most common types of searches is the simple keyword-based search. For this kind of search the query consists only of one or more keywords, without any prefixes and no pattern matches. Such searches are performed only on one index, the main Lucene index containing the information of the crawled classes in the *content* and *comments* fields. Since Lucene is the underlying engine, the Lucene algorithm is used to rank the results based on how many of the query keywords are included and how often. Therefore, the top-ranked result contains the most query keywords and / or the query keywords occurring most frequently. However, the “comment” field has a lower weighting since the source code is normally the main point of interest of the user.

Type 2: Comment-based search

Like keyword-based searches, this kind of search is performed only on the Lucene main index, but limited to the comments field. The actual technical code is not taken into account in this case. This type of search therefore essentially represents a simple type of natural language analysis, and much more sophisticated natural language processing approaches could be included in the future. The popularity of natural language searches is reinforced by studies of the log-files of code search engines which showed that even developers likes to search for code in natural language such as “How to validate a number” to find components which validate a number [BL12a]. However, to include more advanced algorithms it would be also necessary to provide a way for the search engine to detected if natural language processing is desired in a query.

Type 3: Method search

Another more technical type of search is focuses on specific parts of the source code, the methods. The prefix “method:” in a query forces the search engine to explicitly search for methods that match the keywords following the prefix. In contrast to the signature-based search queries of MQL, DAQL method searches do not have to have a specific form like *addCustomer(Customer):void;*, or be embedded in a specific class context. Instead, DAQL allows method searches to be formulated like normal keyword-based searches where the different terms can be written in arbitrary order. This is possible because, in contrast to the previous two searches, method searches are performed on a separate method index. This stores information about different aspects of methods in different fields such as a multi-field for every parameter etc. This has the advantage, compared to the original signature-based search technology of Merobase, that methods can also be matched which do not satisfy all the constraints of the search query such as, for example, methods which have a return parameter that is not specified in the search query. The signature-based search approach requires method signatures in the index to exactly match the search query. In contrast, DAQL allows keywords to appear in any order in the method signature (name, parameter, etc.) and uses special prefixes to define what role they should play in the method. These prefixes include “ip:” for input parameters, “op:” for output parameters or “mn:” for the method name.

Type 4: Interface-based Search

This type of search employs the interface-based search approach implemented in the original version of the Merobase search engine, described by Hummel [Hum08]. Interface-based searches essentially looks for components that realize an interface described in an MQL interface description, including method names and signatures. For this purpose Merobase’s Lucene index contains a set of carefully defined fields which stores the name of the classes as well as each individual method signatures. The search process simply regards the individual method-signatures and the name as “keywords” and perform a search for them in the class index. The key trick is to focus on the fields *name* and *methodSignature*, but with a lower boost for the *name* field as users are usually more interested in the methods than in the name [Hum08] in this type of search. However, Hummel identified several weaknesses of the interface-based search approach in his thesis. One of the weaknesses is that Lucene returns results even when some of the methods contained in the query are not present, so that the user has to perform additional searches. The other weakness is that Lucene does not allow information in the *methodSignature* fields to be tokenized. Again this means that the information in the *methodSignature* field must exactly equal the query for a match to be recognized. This means that acceptable (i.e. relevant) candidates are sometimes missed, for example, if they have an output parameter which is not specified in the search query. Nevertheless, despite these weaknesses, Merobase interface-based searches significantly increased the precision and recall of this type of search compared to other code search engines that existed at the time.

Type 5: Dependency-Aware Search

The final type of search is dependency aware search which is supported only in DAQL and is driven by a slightly different process since additional steps are performed. Particularly as graph-based searches can be quite costly, the first step in a dependency-aware search is a pre-search in the Lucene class-index to determine the best entry-nodes into the graph. This pre-search is a normal interface-based search of the kind just described. However, because DAQL allows searches on multiple classes, the query is first analysed to identify the “central” class – that is, the class with the most outgoing relationships to other classes defined in the “deps:” section of the search query. For example, the query for the *CustomerManagement* system also contains the *Customer* class and the *TaxType* to determine the tax category of the customer.

```
Defs:{    C1:CustomerManagement(
              getCustomer(int):Customer;
              addCustomer(Customer):void;
              udpateCustomer(Customer):Customer;
     );
     C2:Customer(
             getName():String;
     );
     C3:TaxType(
             getTax(Customer):TaxType;
     );
}
Deps:{
             C1→C2; C1→C3
}
```

Here the class *C1* (*CustomerManagement*) has two outgoing relations to the other classes, *C2* (*Customer*) and *C3* (*TaxType*), whereas the other classes have no outgoing relationships. Therefore the class *C1* or *CustomerManagement* is the central class in the query and is used as the basis for an interface-based search on the Lucene class index. The top-ranked results of this search are then taken as entry points in the graph to check if the classes are connected to other classes with a name similar to *Customer* and *TaxType* which contains the specified methods. This process is also repeated for all the other existing relationships specified in the search query. However, if the relationships are extends relationships or method call relationships, this is taken into account for the search of the "central" class in Lucene.

In addition to the different base search types, it is, of course, possible to add additional constraints to search queries such as constraints on the “lang” or “url”. These conditions affects only the number of results, although in our current prototype index “lang” currently has no effect on the result size since it contains only Java source code. Nevertheless, the fields needed to support other programming languages in the future have already been integrated, based on the approach taken in the original version of Merobase,

7.3 Classification using Graph IR Methods

Although this approach could be regarded as a graph IR method, such a classification would not be completely accurate because DAISI does not exclusively use graphs. It also uses text-based Lucene indexes to select the starting points for graph searches and thus to improve the efficiency and effectiveness of the overall process. More specifically, it uses the Lucene indexing approach from Merobase since this was specifically developed to support semantic code searches that are “aware” of the special meaning of the text in source code. Merobase’s Lucene index provides the optimal basis for finding a set of candidate components for deeper analysis using graph search techniques. However, the downside of using Lucene in this way is that its underlying scoring mechanism is not ideal for ranking software components according to their functional relevance. Nevertheless, this is not a big problem in out context because (a) the results from the initial Lucene search are only reduced by the graph-based search (i.e. no new components are ever added) and, (b), nodes with many incoming connections do not necessarily need to have their score boosted. Whereas in general purpose Internet searches incoming relationships are regarded as an indicator of the importance of a web page, in the field of software component retrieval they only provide information about the quality of a component, not about its functionality (and thus relevance).

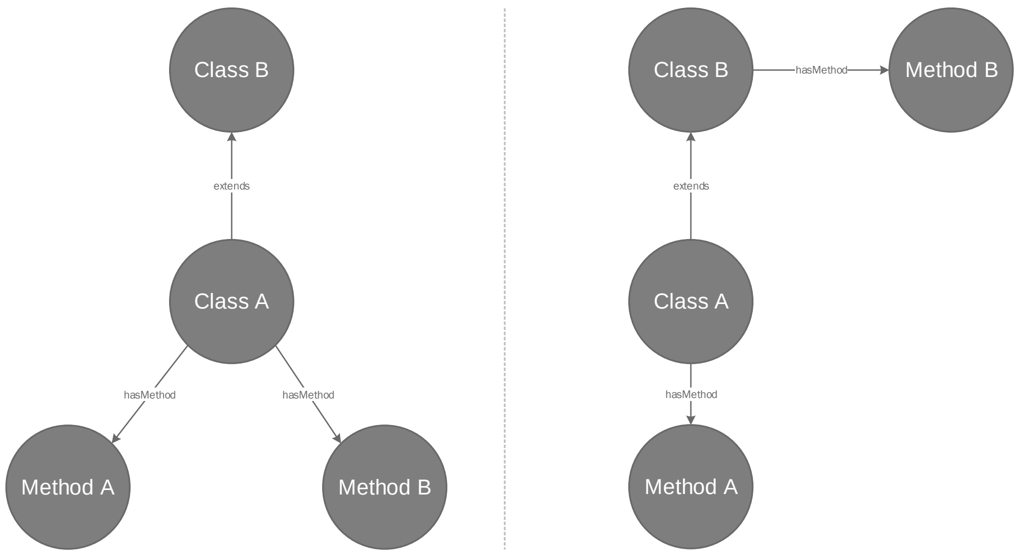


Figure 7.1: Difference of if a method is inherited or not

A direct implementation of the old search mechanism would only reduce the number of results, but as already mentioned, would not improve the precision or recall. This is why the search mechanism implemented in DAISI has been changed minimally, but in a way that has a significant impact on the results. Instead of combining individual conditions using “AND”, as in Merobase (i.e. to required that a result must contain a method A and method B), in DAISI they are combined by “OR”. In other words, a class is only required to contain method A or method B, but not both. At first glance, this might not appear to be a significant change, but the scores of results that contain both methods are boosted by Lucene. It also means that components returned in a search might not directly meet all of the individual conditions specified in a query. While this might be a significant problem for normal Lucene searches, however, in our context it is actually an advantage because it includes components that provide the required features indirectly (e.g. via inheritance). The analysis of the graph determines whether such cases are relevant.

8. Diagrammatic Query Definition

I have discovered that there are two types of command interfaces in the world of computing: good interfaces and user interfaces.

– Daniel J. Bernstein –

In 2009 Brandt et al. came to the conclusion that a modern search engine needs to offer ways of specifying search queries that go beyond traditional, mainstream database systems [Bra+09]. This applies to code search engines as well as normal databases because the complexity of code search queries can grow quickly, especially when relationships between classes are involved. Providing simpler, more efficient ways for users to express queries not only increase the chances that their searches will ultimately be successful, it can also lead to changes in how and when developers search for reusable code within the overall software engineering process.

8.1 The Search Event

The studies of the log-files of two code search engines [PPZ11] [BL12a] mentioned in the previous chapter showed that most search queries were formulated in ways that reflected the implementation problem that users were tackling at the time. Sim et al. also came to the same conclusion in their study [Sim+11], but they also observed that developers do not use search engines in a systematic way at pre-planned points in the development process. Instead, they search for code in totally unplanned ways. Users usually only resort to code search engines in the implementation phase when most of the code already exists and the probability of finding components that can be reused "as-is" is slim [HW07] [HDK06]. At that stage in the development process, the only viable option is usually ad hoc reuse in which the discovered components have to be modified in some way to fit to the existing application. However, modifying code is costly, and can often be more expensive than directly writing the code from scratch [HW08]. As Sommerville observed, since systematic software reuse should be a key ingredient of all software engineering processes, a better way of integrating software search into the development process could have a massive impact, especially at the early stages of development [Som01]. In particular, if software reuse could be integrated seamlessly and performed in a systematic way, many of the current problems caused by ad hoc reuse could be avoided.

8.1.1 Reuse Scenarios

Before discussing how component searches can be integrated seamlessly into early parts of the software development process, in this section we first identify the scenarios and uses cases in which searches are likely to be helpful. Although the question might appear similar to the question "why do developers search for code", developers' underlying motivation is not the only factor that determines how and when they search for code. The overall nature of their specific use cases is also a big factor. For example, a user's motive for performing a search may not be to find directly reusable candidates, but also to find the reason why an exception is thrown in existing code.

In his book [Som01], Sommerville pointed out that code searches can occur at various stages of the development process and presented the outline of a re-use process that takes into account different kinds of re-use. He also observed that the advantages of such a process would be higher reliability, lower risk (i.e. lower uncertainty in new development project), more effective use of specialists, compliance with standards and, as a consequences of all these factors, lower costs. To identify the steps involved, he divided the re-use process into three different categories:

- **reuse of applications:** a complete system is integrated and re-used without adaptation,
- **reuse of components:** individual subsystems or single objects are reused without change,
- **reuse of functions:** components, which contain several functions (e.g. a date converter), are re-used.

Here, the re-use of applications means that a complete software system is integrated into a new system, or the new system is connected to the reused system. This mainly happens for example in the area of databases, since databases are typically stand-alone systems accessed in a loosely coupled way from other systems. However, developers usually make decisions to use databases before the implementation of their own system starts so that they can be sure to create and use the database management system according to its specification.

Component searches can be performed as part of both planned and unplanned reuse. The planned reuse of components occurs, for example, when the decision is made to use third-party libraries, or libraries which were developed in previous projects. Such decisions are typically made during the planning phase, using prior experience to select which existing functionality in self-built libraries can usefully be incorporated into the new system. This can be mainly observed in the context of database connectivity where developers frequently choose to integrate e.g. Hibernate to access databases from their system. Such choices made at the beginning of a project have in a lot of cases direct influence on the architecture and overall structure of the system under development.

In contrast, the unplanned (i.e. ad-hoc) reuse of components occurs when developers come up against an unexpected problem such as converting data from one format to another. In such cases, they will probably either search for source code to parse the data, or search for existing libraries that provide the required functionality.

The reuse of functionality almost always takes place in an unplanned way since at this level, searches primarily focus on snippets of source code. This usually happens during the implementation phase when developers address specific coding challenges, like the one above.

As well as identifying these categories, Sommerville also specified the criteria that need to be fulfilled in all three case to make reuse possible -

- it must be possible to search for appropriate components which have to be categorized correctly,
- users of re-useable components must be convinced they behave as indicated and are reliable,
- the components need to be well documented so that developers can understand their functionality.

In addition to these criteria, Sommerville also identified several issues which can arise during the process of re-use. One of the major potential problems he identified are maintenance costs because if a component's source code is not written by the developer himself, maintenance becomes significantly more difficult. The side effects that occur due to changes in the new code can sometimes be particularly hard to detect. Theoretically, a public available library should be maintained by the developers, but unfortunately this is the exception rather than the rule. There are many examples of open source libraries offering very useful features which are no longer maintained. This, of course, raises significant doubts about the quality of a library in the minds of developers, and often causes them to avoid a component even though it offers precisely the functionality they need.

Another issue Sommerville identified is the presence of large gaps in the availability of tools that support re-use. This issue has been alleviated to a large extent since 2001 by the release of many different code search engines and recommendation tools as IDE plug-ins.

Finally, the last issue identified by Sommerville is the psyche of the developers and the importance of the "not-here-invented" syndrome [All+88]. Many developers do not trust the skills of other developers and prefer to develop all parts of their systems themselves, especially smaller components, regardless of the time they need to develop and maintain them [Som01].

In contrast to Sommerville, who defined his categories of re-use scenarios in terms of the size of components, a few years before Sim et al. defined 11 different types of searches [SCH98]:

1. searches for all uses of a variable or function to assist in impact analysis,
2. searches for function and variable definitions to assist in program understanding,
3. searches to reuse functions, variables or objects,
4. searches for function signatures to call them correctly,

5. searches for functionality that is known to exist, but where the name may not be known,
6. searches identify misbehaving code for maintenance (i.e. bug location),
7. searches to track the usage of a variable,
8. searches to find an output string as the starting point for a bug hunt,
9. searches to find all uses of an entity being removed to eliminate dead code
10. searches to analyse variables when porting code,
11. searches to examine functions when adding new features.

Most of these types focus on local searches, since at the time Internet-based search over the web or distributed repositories was only just emerging. In contrast, in 2008 Umarji et al. reduced this set of search types to just 9. However, they ignored local searches, so all their search types focus on public available repositories [USL08]. Umraji et al. also introduced the idea that types should be characterized by two dimensions – motivation and size. The motivation dimension addresses whether a search is being performed to support the "as-is" reuse of components, to find reference examples, or to find bugs in code. As its name implies, the size dimension addresses the magnitude of the reused artefact, and is subdivided into three further subcategories, code-blocks (e.g. like wrappers or parsers) subsystems (e.g. like algorithms or libraries) and systems (e.g. complete stand-alone runnable applications).

In terms of "as-is" reuse, in the motivation dimension Umarji et al. identified 4 different types of artefacts as the subject of searches:

1. code snippets, wrappers or parsers,
2. data structures, algorithms and GUI widgets to be incorporated into an implementation,
3. libraries to be incorporated into an implementation,
4. systems to be used as the starting point for an implementation.

In terms of the "find reference" scenario in the motivation dimension, they also identified 4 different types of artefacts as the subject of searches:

1. blocks of code to be used as an example,
2. examples of how to implement a data structure, algorithm or GUI widget,
3. examples of how to use a library,
4. similar systems to be used as a source of ideas.

Only one of Umarji et al.'s search types, the ninth "Confirmation and resolution of defects", is not related to the "as-is" reuse or "reference example" motivations. Instead, this type focuses on the process of detecting bugs in a system and establishing the meaning of the error messages. A closer look at Umarji et al.'s search types reveals that only the first two types in each dimension occur when developers are currently facing a problem. Such cases, of the kind identified by Stolee et al.'s study, correspond to true classic reuse scenarios [SED14] and to a certain extent are already supported by today's search engines. The other types involve much more complex search scenarios, such as exploiting the relationships between different classes and are basically not supported today. Only searches for existing libraries are more or less covered by the big, mainstream search engines like Google, Bing or Yahoo, as long as they are well documented so that they can be discovered using normal keyword-based searches. However, such documentation normally only exists for the larger and more widely used libraries.

The search types related to the discovery of existing data structures are currently very limited since no existing search engine supports a sufficiently powerful query language for specifying the required relationships and/or structures. Consider, for example, the second "how to implement a data structure" in the list of search types described above. This is a case where, in general, several preliminary steps have to be performed such as (1) analysing the requirement and extracting the domain objects, (2) creating a diagram describing the relationships between these domain objects,

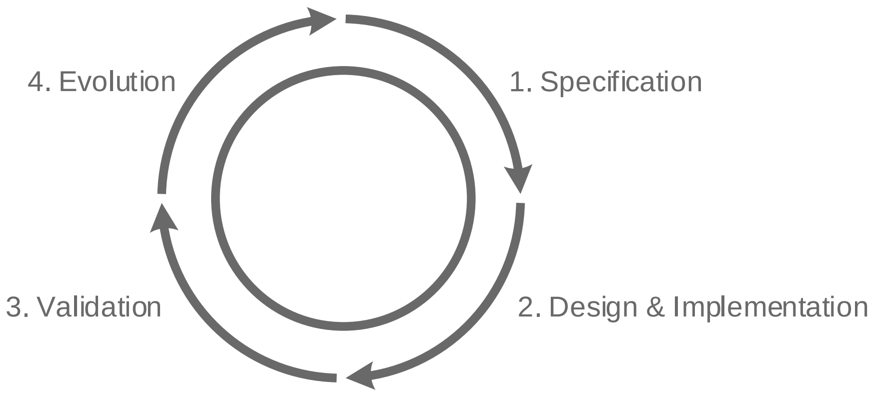


Figure 8.1: Design -> Implementation -> Validation Process

(3) adding any necessary technical classes to the class diagram, (4) implementing the classes and (5) validating the implementation. Such steps are performed more or less universally regardless of the underlying process, like the waterfall process, the RUP or another process. The only difference between such processes is the amount of time spent on each of these steps. Also, in the waterfall process the steps occur only once, whereas in agile processes these steps are performed in several iterations. However, the *Design*, *Implementation* and *Validation* steps are always performed at some point.

Searches for existing code components typically occur during the implementation phase, when developers are aware of the structure, names and the functionality of the parts of the system from design documents such as UML diagrams. Therefore, rather than addressing the question of "how to implement a data structure" directly by building the parts from scratch, it would be helpful if suitable implementations could be found using the information in the UML diagram directly. As well as saving the time that would be needed to implement the component from scratch, such a capability could also reduce the problem of component integration since the accompanying development effort can focus on adapting discovered components.

Moreover, rather than performing such code searches in the implementation phase they could be performed at the end of the design phase when the UML design diagrams are created. This means that the output of the design phase would not only be a design in the classic sense (with the accompanying documentation and models), but also a collection of candidate components for implementing the system. Such an approach would also fit well into agile processes where the different steps are performed several times, since designs are created at different levels of granularity, throughout the process.

The biggest challenge to provide such a search capability is to ensure the relevance of search results. Since the results of such searches will often be groups of classes, rather than individual classes, it is not only essential that the individual members of the group are sound and of high quality, it is also essential that they fit together correctly to meet the user's needs. The time saving benefits of this kind of search significantly decreases if the developer has to search for every single class individually and has to adapt and connect all the discovered classes to each other manually. The next section therefore presents a new type of search capability, driven by diagrammatic (i.e. UML-based) queries, which considers inter-class relationships to deliver groups of components that satisfy the user's needs.

8.2 UML-based Search

The UML is the most widely used modelling language for describing and visualizing designs of software systems. It is commonly used before any line of code is written to model and plan the structure and architecture of a software system. Thanks to its ability to display almost all processes and structures within software systems in a programming-language independent way, the UML is used in all kinds of software development processes. UML diagrams therefore provide the ideal input for searching for complex software structures and support reuse at a higher level of granularity than existing search engines. Nevertheless, to support UML-based diagrammatic search queries it is necessary to transform UML diagrams into the DAQL query format described in chapter 7.

Although the UML visual syntax is standardized, this is not the case for the underlying file format. As a result, many of the underlying file formats used by different UML tools are incompatible. One of the most widely used and freely available formats is the EMF (Eclipse Modeling Framework) file format defined by the Eclipse Foundation. The prototype UML search capabilities developed in this thesis therefore uses the EMF format to describe UML-based queries. Only UML class diagrams are supported since these contain all the information that can currently be used to perform searches, like the extends, realizes or uses relationships between classes and interfaces. Other UML diagram types such as sequence diagrams also contains useful information about methods, but since the goal of this thesis is to exploit structural information, the current focus is on class diagrams. Most open source UML modelling tools, such as Papyrus or UML Designer support the EMF format. Therefore, it is possible to create UML queries in a conventional UML modelling tool and export them to the DAISI search engine.

To convert the EMF representation of a class diagram to a DAQL query, the different elements of the UML diagram need to be mapped to different parts of a query. The first step is to extract classes from the diagram because classes are the most important elements. Since classes and similar concepts like enums or interfaces are represented in the EMF data structure as “eClassifier” elements this can easily be achieved by simply collecting all instances of “eClassifier”. The specific kind of classifier represented by an “eClassifier” element is stored in the attribute “xsi:type”. For example, a class has the type `ecore:EClass`. Interfaces are not designated by this attribute, however, but by an additional Boolean “interface” attribute which has the value “true” for interfaces and “false” otherwise. While the different kinds of classes are being extracted, a counter is internally incremented to facilitate the creation of unique DAQL keywords.

```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore"
      name="rootElement" nsURI="http:///rootElement.ecore"
      nsPrefix="rootElement">
   <eAnnotations source="http://www.eclipse.org/uml2/2.0.0/UML">
      <details key="originalName" value="RootElement"/>
   </eAnnotations>
   <eClassifiers xsi:type="ecore:EClass" name="CustomerManagement">
      <eOperations name="addCustomer" ordered="false" lowerBound="1">
        <eParameters name="customer" ordered="false" lowerBound="1"
               eType="#//Customer"/>
      </eOperations>
      <eStructuralFeatures xsi:type="ecore:EReference" name="customer"
                      ordered="false"lowerBound="1"
                      eType="#//Customer">
      </eStructuralFeatures>
   </eClassifiers>
   <eClassifiers xsi:type="ecore:EClass" name="Management"/>
   <eClassifiers xsi:type="ecore:EClass" name="Customer">
      <eOperations name="getName" ordered="false" lowerBound="1">
      <eParameters name="name" ordered="false" lowerBound="1"
               eType="ecore:EDataType
               http://www.eclipse.org/emf/2002/Ecore#//EString"/>
      </eOperations>
   </eClassifiers>
 </ecore:EPackage>
```

Listing 8.1: ecore model of the CustomerManagement example

More specifically, every element is internally identified by *C1* ... *CN* identifiers for classes or *I1* ... *IN* identifiers for interfaces. In the next step, the individual methods are extracted from each class. This is achieved by extracting all sub-elements of type "eOperations". These elements themselves have sub-elements of type "eParameters" which represent the input and output parameters. For our purpose, only the types of the parameters stored in the `eType` attribute are used.

Once all information about the individual classes has been extracted from the UML diagram, the final step is to extract the information about the inter-class relationships. This information can be found in the "eStructuralFeatures" elements of type "ecore:eReference" and is used to build the "deps:" section of the corresponding DAQL query. No such element exists for interfaces, however, since interfaces as well as superclasses are both referenced via "eSuperTypes" elements. To extract the relationship information, therefore, it is necessary to look at the type of the "superclass". If it is a normal class, the relationship between the two classes is "extends", whereas if it is an interface the relationship is "realizes".

```
Defs:{    C1:CustomerManagement(
          addCustomer(Customer):void;
    );
    C2:Customer(
         getName():String;
    );
    I1:Management()
}
Deps:{
         C1→C2; C1--→I1
}
```

As can be seen from the above example, the result is a DAQL query which can be used to drive a dependency-aware search via DAISI. The only problem that can occur

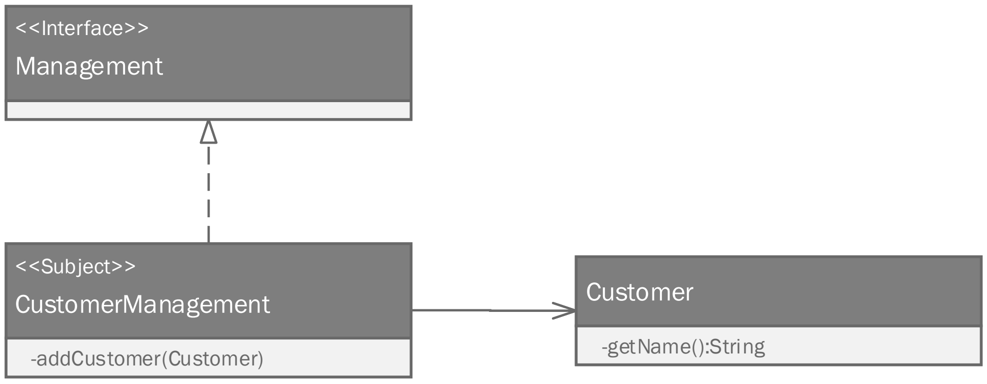


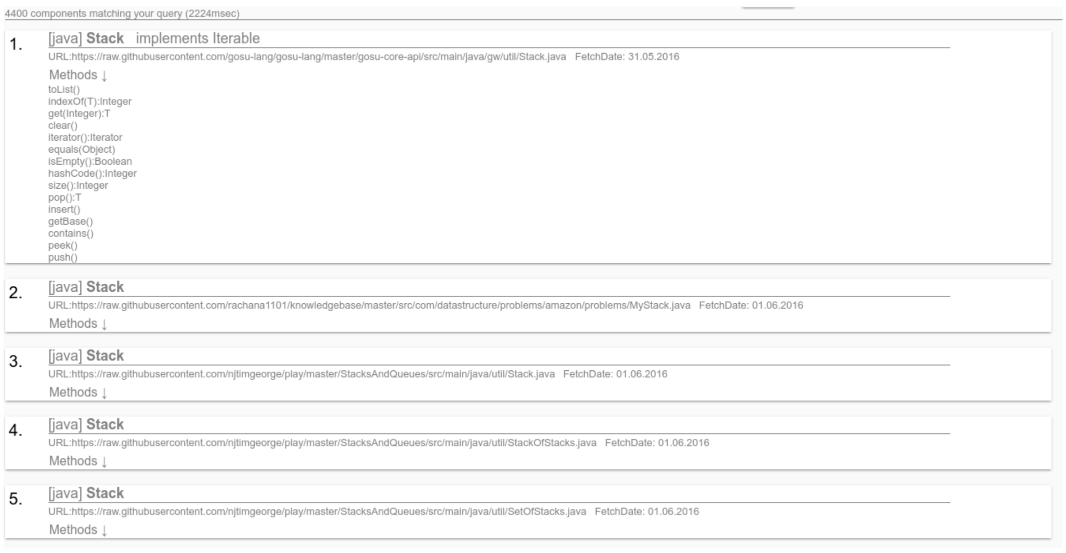
Figure 8.2: KobrA representation of the CustomerManagement example

with this kind of search is the size of the UML diagram. The larger the UML diagram the greater the search time and the greater the likelihood that the focus of the search (i.e. the class the user is primarily interested in) could be mistaken. This means that the initial Lucene search will be performed based on the “wrong” class, from the perspective of the developer. A strategy for tackling this problem would be to use a methodology such as KobrA [Atk+08] which explicitly calls for the identification of the subject class of all class diagrams, and explicitly advocates a modelling approach where all the information in a model revolves around that individual subject.

8.3 Search User Interface

To provide a simple and intuitive interface to the dependency-aware search capabilities as part of this thesis a Web 2.0 front-end was developed.

Like other search engines, the front-end minimizes obfuscating information on the starting page. Therefore, the homepage only contains a text box where the query can be entered, a help button to receive information about the query language and a bar along the top, listing the different types of search that can be performed. These categories are normal DAQL text-based searches, UML-based searches of the kind described in the previous section, and special “drag an drop” search which will be

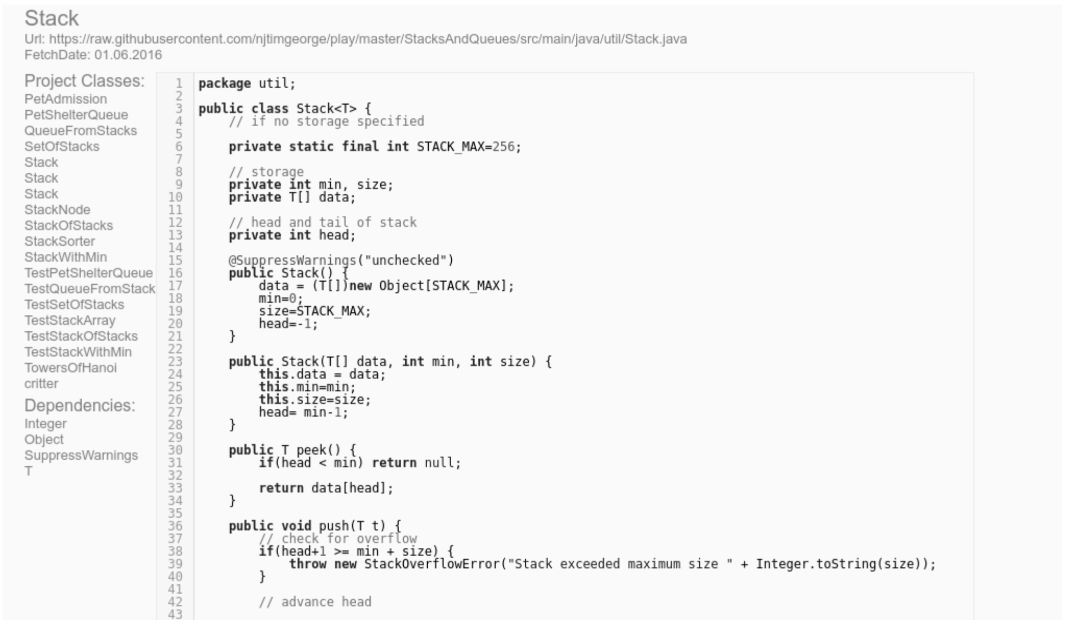


```
4400 components matching your query (2224msec)

1.  [java] Stack   implements Iterable
    URL:https://raw.githubusercontent.com/gosu-lang/gosu-lang/master/gosu-core-api/src/main/java/gw/util/Stack.java   FetchDate: 31.05.2016
    Methods ↓
    toList()
    indexOf(T):Integer
    get(Integer):T
    clear()
    iterator():Iterator
    equals(Object)
    isEmpty():Boolean
    hashCode():Integer
    size():Integer
    pop():T
    insert()
    getBase()
    contains()
    peek()
    push()

2.  [java] Stack
    URL:https://raw.githubusercontent.com/rachana1101/knowledgebase/master/src/com/datastructure/problems/amazon/problems/MyStack.java   FetchDate: 01.06.2016
    Methods ↓

3.  [java] Stack
    URL:https://raw.githubusercontent.com/njtimgeorge/play/master/StacksAndQueues/src/main/java/util/Stack.java   FetchDate: 01.06.2016
    Methods ↓

4.  [java] Stack
    URL:https://raw.githubusercontent.com/njtimgeorge/play/master/StacksAndQueues/src/main/java/util/StackOfStacks.java   FetchDate: 01.06.2016
    Methods ↓

5.  [java] Stack
    URL:https://raw.githubusercontent.com/njtimgeorge/play/master/StacksAndQueues/src/main/java/util/SetOfStacks.java   FetchDate: 01.06.2016
    Methods ↓
```

Figure 8.3: Search result list

described later in this section. The search results are presented in the standard style typical for other code search engines. The default is to show only the first 10 results on the first page. However, to help to decide which results are worthwhile examining in further detailed, the result list provides additional information along with the name of the components – the programming language, in front of the name, the URL where the component was found and the methods offered by the component. To reduce the used space and to provide the user with an overview, the methods are initially hidden, but can easily be called up on demand as shown for the first result in figure 8.3.

After selecting a result, the user is forwarded to a page presenting the details of the corresponding component. As shown in figure 8.4, the source code takes centre stage in the detailed view since this is the ultimate description of the functionality offered by the component. However, on the left hand side, it is possible to search for dependencies as well since all the included classes in the project from which the component was obtained. Thus, if a user wishes to see what classes the component is dependent upon, these are directly accessible from the details page.



Stack

Url: https://raw.githubusercontent.com/njtimgeorge/play/master/StacksAndQueues/src/main/java/util/Stack.java

FetchDate: 01.06.2016

Project Classes:

PetAdmission
PetShelterQueue
QueueFromStacks
SetOfStacks
Stack
Stack
Stack
StackNode
StackOfStacks
StackSorter
StackWithMin
TestPetShelterQueue
TestQueueFromStack
TestSetOfStacks
TestStackArray
TestStackOfStacks
TestStackWithMin
TowersOfHanoi
critter

Dependencies:

Integer
Object
SuppressWarnings
T

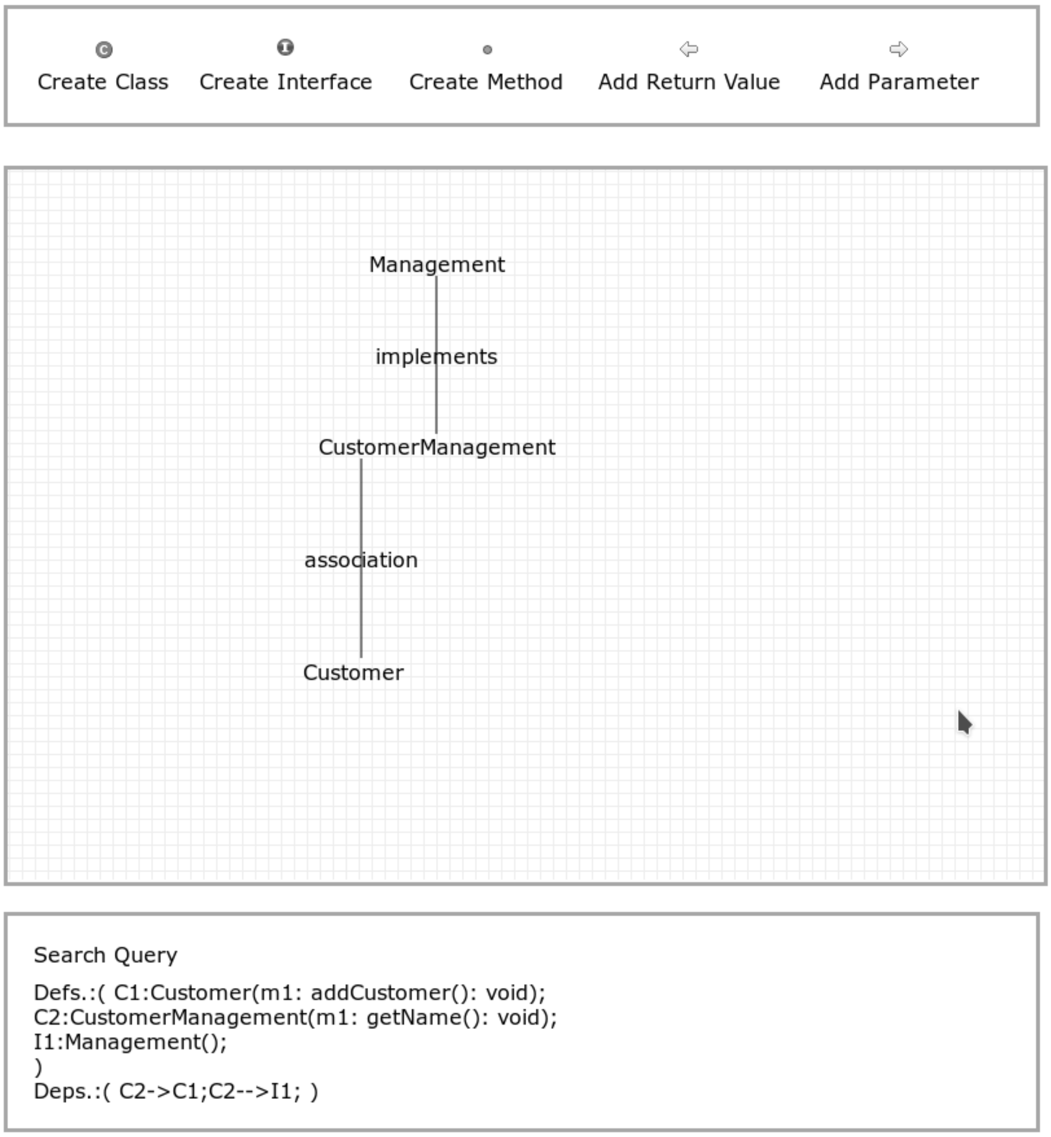
```
package util;

public class Stack<T> {
    // if no storage specified

    private static final int STACK_MAX=256;

    // storage
    private int min, size;
    private T[] data;

    // head and tail of stack
    private int head;

    @SuppressWarnings("unchecked")
    public Stack() {
        data = (T[])new Object[STACK_MAX];
        min=0;
        size=STACK_MAX;
        head=-1;
    }

    public Stack(T[] data, int min, int size) {
        this.data = data;
        this.min=min;
        this.size=size;
        head= min-1;
    }

    public T peek() {
        if(head < min) return null;

        return data[head];
    }

    public void push(T t) {
        // check for overflow
        if(head+1 >= min + size) {
            throw new StackOverflowError("Stack exceeded maximum size " + Integer.toString(size));
        }

        // advance head

```

Figure 8.4: Details of a component

8.4 Drag and Drop Search

In addition to the text-based and the UML-based search interface, DAISI offers another way of defining search queries. This interface is mainly aimed for developers who do not want to write a complex textual search query, but also do not want to create a UML diagram. It should also be useful to developers who only want to search for components in part of an existing UML diagram, but would like to avoid having to explicitly create a separate UML diagram with just the information they are interested in for the search. The third interface therefore allows queries to be assembled using a simple “drag and drop” metaphor. Using this interface, users can draw the different classes and associations in the typical UML style, but without the full power (and associated overhead) of the full UML.

As shown in figure 8.5, it is possible to drag the elements in the bar at the top directly into the drawing area in the middle to assemble a query. Associations can be created by clicking on an element and dragging the association type that appear in the resulting dialog box to the other element to which the relationships needs to be connected.



Search Query
```
Defs.:( C1:Customer(m1: addCustomer(): void);
C2:CustomerManagement(m1: getName(): void);
I1:Management();
)
Deps.:( C2->C1;C2-->I1; )
```

Figure 8.5: UI of the “drag and drop” search possibility

These three ways of specifying queries cover all the usual preference of developers. As with other code search engines, the most expressive way of writing queries is using the textual language (i.e. DAQL). However, because such queries can quickly become quite complex, there is a risk that DAISI will face the same problem as other search engines in which the majority of developers fail to tap its full potential. Since powerful features that are rarely if ever used will not make a significant contribution to software reuse in the long term, the other two more user-friendly search interfaces make it possible for developers to use these powerful features in more intuitive ways. This is particularly helpful when multiple relationships needs to be specified in an unplanned ad-hoc search. With this “drag and drop” user interface developers can avoid having to learn the technical details of the syntaxes of UML and the underlying query language.

9. Evaluation

Everything that can be counted does not necessarily count; everything that counts cannot necessarily be counted.

– Albert Einstein –

Any new approach must be critically appraised to evaluate whether it delivers the claimed benefits. As indicated by the hypotheses list in the introduction, in the field of information retrieval this largely comes down to demonstrating improvements in the precision and recall. In this chapter we presents the results of an evaluation that compares the recall and precision achieved by the new technology, developed embodied in the DAISI prototype, to Merobase using the metrics precision@10 (P@10) and precision@5 (P@5) [Ort+16]. These metrics evaluate the occurrence of relevant results in the top 10 and 5 results returned by a search respectively. However, since it is difficult to compare an approach that is able to handle inter-component relationships to an approach that is not, the evaluation first compares their performance on "traditional" search scenarios where Merobase is able to deliver appropriate results. Merobase is chosen for the comparison not just because the new technology subsumes its query language but because other leading search engines of that generation, like Portfolio or Sourcerer, are no longer available. To perform a fair comparison of the underlying technologies, and not of different repositories, the same data-set was used to compare their performance. For this purpose we constructed a repository and an index containing about 500.000 different classes and 1.5 million methods from 30.000 projects, mainly crawled and parsed from GitHub. This common data-set was used to compare several search examples and realistic scenarios which frequently occur during software engineering projects.

9.1 Simple Case

The "classic" example used by Hummel to show the benefits of interface-based search is a *Stack* containing a *pop()* and a *push()* method. Of course, this is a really simple example and even if the discovered components can not be reused "as-is", an experienced developer can normally adapt classes of this scale with ease. Nevertheless, although Merobase's interface-based search queries allowed users to specify what they are looking for much more precisely than previous search engines they are still rather generic. For example MQL does not allow a developer to specify that he/she is not interested in any kind of stack, but rather stacks storing a specific type of item without using the Java generic types mechanism. The developer might desire functionality to sort the contained items, too, which is much more likely to be included in a stack of `Item` objects. However, for this scenario it is necessary to consider two different use cases. In the first use case the user has already obtained or developed a class `Item` and is looking for a *Stack* class to store instances of it. More specifically, this use case involves –

> a search for a `Stack` class which is specifically designing to store objects of an existing class called `Item`.

In the second use case, however, the user does not yet have the class of the object to be stored in the stack, but knows that it should have an attribute to store a name and a method to access this attribute. This case, therefore, involves –

> a search for a class, `Stack`, designed to store objects of a class which has an attribute to store a name and a method to get the value of this attribute.

9.1.1 Case 1

The first scenario can be handled by the Merobase signature-based search technology and delivers quite reasonable results since it is possible to define a query for a class named `Stack` with `push` and `pop` methods having a class called *Item* as their input and return parameters respectively.

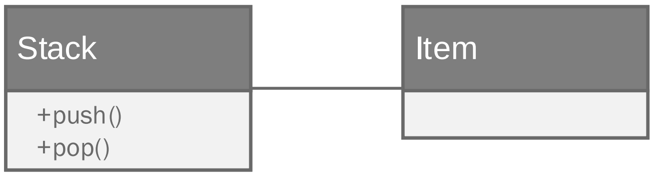


Figure 9.1: *Stack* - *Item* simple example

The corresponding MQL query has the following form:

```
Stack(pop():Item; push(Item));
```

On the repository used for the evaluation, this search delivers 80 results. However, closer analysis revealed that only the first three results fully match the requirements of the user, as expressed in the query, and of these, only the first two can be used “as-is” without modification. The third result can be made to match the requirements with minor adaptations and the fourth and sixth can be used as a reference example.

However, the fifth result, which manages Item classes, too, does not manages the Items in a way of a Stack and therefore do not fulfil the search requirements of the query

1. Stack

   Methods: pop():Item, push(Item), size(), iterator(), main()

   URL: `http://www.cs.princeton.edu/courses/archive/fall17/cos126/precepts/Stack.java?highlight=off`

2. Stack

   Methods: pop():Item, push(Item), size(), iterator()

   URL: `https://raw.githubusercontent.com/amitbansalite/Coursera_Algortihms_Princeton/master/algs4/Stack.java`

3. RezisingArrayStack

   Methods: pop():Item, push(Item), iterator()

   URL: `https://raw.githubusercontent.com/amitbansalite/Coursera_Algortihms_Princeton/master/algs4/ResizingArrayStack.java`

4. Lifo

   Methods: pop():Item, getSize(), setSize(int), getPremierItem(), setPremierItem(Item)

   URL: `https://raw.githubusercontent.com/joanny/T_P/master/Crytographie/src/com/iut/LIFO/Lifo.java`

5. Indicator

   Methods: setUrl(Item), getUrl():Item, getCall():Item, setService(Item)

   URL: `https://raw.githubusercontent.com/eschwabe/interview-practice/master/coursera/algorithms-part1/stacks-and-queues/StackWithMax.java`

6. ViewRequestController

   Methods: no relevant

   URL: `https://raw.githubusercontent.com/hartmannr76/DogEBooks/master/src/java/controller/requests/ViewRequestController.java`

7. ItemController

   Methods: setItem(Item), getItem():Item, getItems(), find()

   URL: `https://raw.githubusercontent.com/grantbachman/coursera/master/algorithms_1/QueueDeque/Deque.java`

8. Deque

   Methods: addLast(Item), removeLast():Item, addFirst(Item), isEmpty():Boolean, main()

   URL: `https://raw.githubusercontent.com/grantbachman/coursera/master/algorithms_1/QueueDeque/Deque.java`

The repository actually contains three additional classes which satisfy the constraints specified in the search query. The only reason why they were not included is because one or more of the methods has a return parameter. This reveals one of the weaknesses of the interface-based search mechanism in Merobase resulting from the non-tokenized fields within the Lucene index. As already mentioned, if the

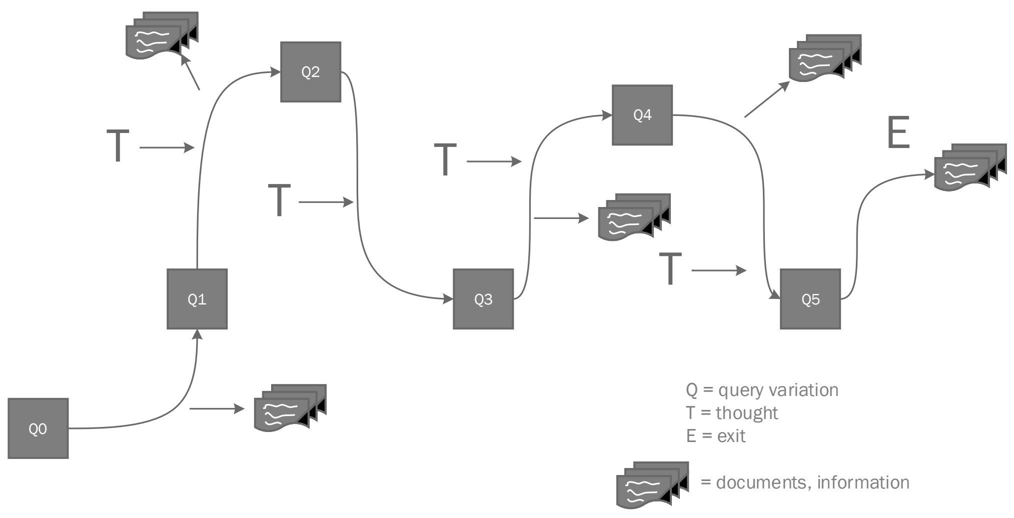


Figure 9.2: Berrypicking search process [Bat89]

specification of a method in the search query does not fully match the signature stored in the index no match will be detected by Lucene and the corresponding component will not be added to the result list. To retrieve these additional components using interface-based search, it is necessary to change the search query slightly and explicitly include the return parameters. Possible options are `boolean`, where the `push` method returns "true" if the item is successfully added to the stack and `Item` where the item itself is returned unchanged by the method. So, for example, the query could be changed to –

```
Stack(pop():Item; push(Item):boolean);
```

This search query, however, does not include the results of the first search shown previously. Thus, with signature based search users often have to apply a Berrypicking process [Bat89] as shown in 9.2, where they repeatedly modify the search query until they obtain acceptable results.

In contrast, the same search, performed as a dependency-aware search in the graph database delivers all the original results, but includes the three relevant classes not included in the first search. This is because the new graph-based search engine does not have this weakness of interface-based search.

1. SLStack

   Methods: push(Item):Boolean, pop():Item, peek():Item, isEmpty(), isFull()

   URL: `https://raw.githubusercontent.com/jedwardblack/DataStructures/master/StackADT/src/SLStack.java`

2. Stack

   Methods: pop():Item, push(Item), size(), iterator(), main()

   URL: `http://www.cs.princeton.edu/courses/archive/fall17/cos126/precepts/Stack.java?highlight=off`

3. Stack

   Methods: pop():Item, push(Item), size(), iterator()

   URL: `https://raw.githubusercontent.com/amitbansalite/Coursera_Algortihms_Princeton/master/algs4/Stack.java`

4. StackWithMax

   Methods: pop():Item, push(Item), max():Item

   URL: `https://raw.githubusercontent.com/eschwabe/interview-practice/master/coursera/algorithms-part1/stacks-and-queues/StackWithMax.java`

5. Lifo

   Methods: pop():Item, getSize(), setSize(int), getPremierItem(), setPremierItem(Item)

   URL: `https://raw.githubusercontent.com/joanny/T_P/master/Crytographie/src/com/iut/LIFO/Lifo.java`

6. ItemController

   Methods: getItem():Item, setItem(Item), isMixedItem(), searchItem(String)

   URL: `https://raw.githubusercontent.com/marembo2008/seamlesspos/master/src/main/java/com/seamless/internal/controller/ItemController.java`

7. ResizingArrayStack

   Methods: pop():Item, push(Item), resize(Integer)

   URL: `https://raw.githubusercontent.com/amitbansalite/Coursera_Algortihms_Princeton/master/algs4/ResizingArrayStack.java`

8. Stack

   Methods: pop():Object, push(Object), ensureCapacity()

   URL: `https://raw.githubusercontent.com/xingyuli/swordess-toy-effectivejava/master/src/main/java/org/swordess/toy/effectivejava/chapter5/use_generic_type_first/Stack.java`

In the above listing of results, the three additional `Stack` classes are the third (`Stack`), the fifth (`StackWithMax`) and the seventh (`SLStack`) result.

Table 9.1 compares the P@10 and P@5 metrics for the two different types of searches.

MQL	
P@5	P@10
0.8	0.4

DAQL	
P@5	P@10
1.0	0.6

Table 9.1: P@5 and P@10 metric of the simple Stack search

This shows that the DAQL queries have much higher P@10 and P@5 scores than the MQL interface based search. Of course, it must also be considered that, compared to other software search indexes, we only have a relatively small index as basis. With a larger index and thus a higher probability of matching results, the Merobase would deliver significantly more and better results.

9.1.2 Case 2

In the second use case, the user wants to find both classes, a `Stack` class as well as an `Item` class with certain characteristics. In MQL the user had to analyse all results of a search for the `Stack` to establish whether one of them uses a class `Item` with the appropriate properties. Alternatively, a user could perform two searches, one for each class. However, based on the search result above, the user would not be able to find a component with these characteristics. To obtain matching results which are connected to an `Item` class containing a name attribute it is necessary to change the MQL query by adding either the `boolean` or the `Item` return parameter to the `push` method .

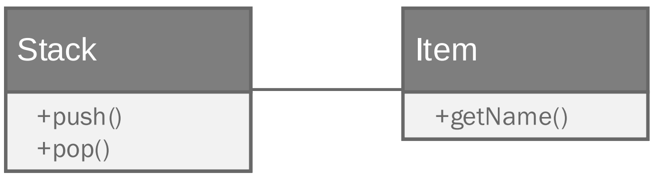


Figure 9.3: *Stack* - *Item* example

To perform the same search using a DAQL dependency aware search, it is only necessary to change the search query to the following –

```
Defs:{C1:Stack(pop():Item;push(Item));C2:Item(getName():String)}

Deps:{C1->C2}
```

This combines the desired constraints on both classes into one search query that find a perfectly matching pair. The probability that a pair fits together increases as the number of properties which are required to have increases. For example, in addition to `pop()` and `push()` methods the `Stack` class might be required to have other functionality such as sorting, conciliation or similar methods.

1. Stack

   Methods: pop():Item, push(Item), size(), iterator(), main()

   URL: `http://www.cs.princeton.edu/courses/archive/fall17/cos126/precepts/Stack.java?highlight=off`

2. Stack

   Methods: pop():Item, push(Item), size(), iterator()

   URL: `https://raw.githubusercontent.com/amitbansalite/Coursera_Algortihms_Princeton/master/algs4/Stack.java`

3. ResizingArrayStack

   Methods: pop():Item, push(Item), resize(Integer)

   URL: `https://raw.githubusercontent.com/amitbansalite/Coursera_Algortihms_Princeton/master/algs4/ResizingArrayStack.java`

4. Lifo

   Methods: pop():Item, getSize(), setSize(int), getPremierItem(), setPremierItem(Item)

   URL: `https://raw.githubusercontent.com/joanny/T_P/master/Crytographie/src/com/iut/LIFO/Lifo.java`

4. StackWithMax

   Methods: pop():Item, push(Item), max():Item

   URL: `https://raw.githubusercontent.com/eschwabe/interview-practice/master/coursera/algorithms-part1/stacks-and-queues/StackWithMax.java`

...

8. SLStack

   Methods: push(Item):Boolean, pop():Item, peek():Item, isEmpty(), isFull()

   URL: `https://raw.githubusercontent.com/jedwardblack/DataStructures/master/StackADT/src/SLStack.java`

The rest of the 10 results are not relevant to the search query, but some of them are `Item` classes which are maybe useful for the developer depending on his/her use case. In contrast, the three results containing `Stack` classes totally match the search query and thus provide software components containing a stack storing items that contains a name attribute and an appropriate getter-method. These results also directly show one of the main advantages of dependency-aware searches – the three relevant results involve the same `Item` class.

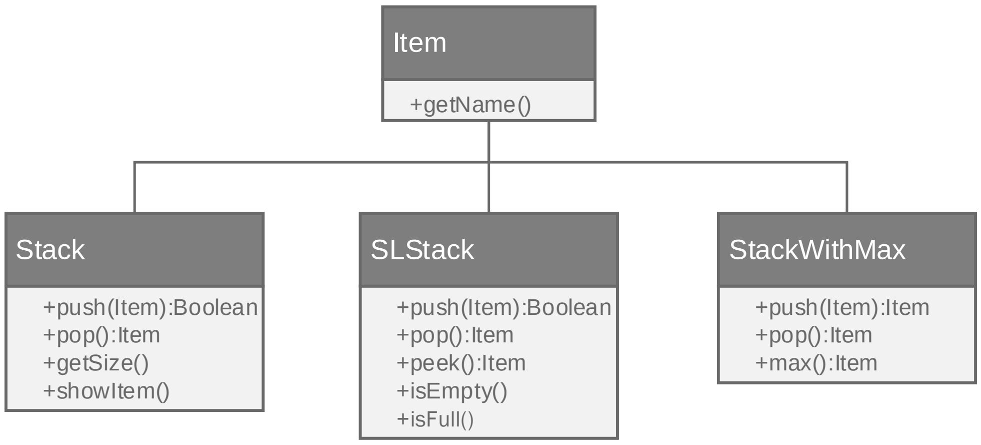


Figure 9.4: Three *Stack* classes connected to the same *Item* class

An analysis of the "projects" of the individual results reveals that the projects from which two of the results were harvested do not contain an `Item` class, because they just contain example classes used for teaching purposes. In fact, they are not complete projects at all. The pairs in the results set were therefore associated with one another during the dependency resolution process.

Although at first sight it might seem meaningless to compare the P@10 and P@5 values of DAISI and Merobase because the latter does not support dependency aware searches, since it is possible for developers to "simulate" dependency-aware searches through multiple signature-based searches, it is possible to compare the two search engines. Table 9.2 shows the P@5 and P@10 values for the single DAQL dependency-aware search compared to the combined results of the two MQL signature-based searches (i.e. the one with return parameters and the one without return parameters).

MQL	
P@5	P@10
0.3	0.1

DAQL	
P@5	P@10
0.8	0.6

Table 9.2: P@5 and P@10 measurement values for the Stack search with matching Item class

Even this simple example shows the benefits of DAISI's dependency-aware search. Although it is possible to perform multiple, reformulated searches in the old MQL search to get appropriate results, this involves significant extra work. The user either has to try all possible return parameter variants to get matching results, or has to take one of the stacks from the first search and adapt it. Dependency aware search increases the precision of searches as well as their recall. The traditional MQL signature based searches returned 34 results, but only 3 of them were relevant to the search query, whereas the new DAQL dependence-aware search returned 10 results in total but 6 of them were relevant. In other words, the new technology delivers more relevant results and far fewer irrelevant results. The precision, recall and the F-measures are defined mathematically as follows:

$$precision = \frac{\#relevant\ retrieved\ results\ in\ the\ index}{\#retrieved\ results}$$

$$recall = \frac{\#relevant\ retrieved\ results\ in\ the\ index}{\#existing\ results\ in\ the\ index}$$

$$F-measure = \frac{2\ x\ precision\ x\ recall}{precision + recall}$$

The F-measure metric is often used to provide a summary as it is the harmonic mean of the precision and recall. It indicates whether an increase in recall outweighs a decrease in precision and vice versa.

	MQL	DAQL
precision	0.088	0.6
recall	0.5	1.0
F-measure	0,1279	0.75

Table 9.3: precision, recall and F-measurement values

The only weakness that can be observed in the current DAISI implementation of dependency-aware search is related to the scoring of components and therefore with the order in which they are represented in the result set. This is determined by the order in which they appear in the normal Lucene search that represents the first step of the search process. Once the initial set of candidates have been found by Lucene, subsequent steps only removes results from the list if they do not match the constraints in the search query. One way of addressing this weakness is to recalculate the scores for each candidate after the graphed-based search and reorder the retained results. Nevertheless, even without such enhancements all examples in this chapter demonstrates that matching results are better and are always returned within the first ten results. Since users usually check at least the first ten results before starting to reformulates queries, this level of performance is usually sufficient [SJC00].

9.2 Methods from Superclasses

To compare the performance of DAISI to Merobase on a non-trivial, yet still simple, example we consider a search for a class which either directly has a particular method or inherits the method from another class. As before, this kind of search can only be performed on Merobase with some extra effort by the users. Nevertheless, this is sufficient to support a comparison between the technologies underpinning the two search engines. The precise example involves –

a search for a class `Customer`, containing a name and a database-created ID, which ideally extends (i.e. inherits from) a class that handles all the database related issues.

A simple keyword search performed on the evaluation repository using the query "customer" returns 57444 results, while a signature-based search using the query

```
Customer(getName():String;getId(); setName(String))
```

returns 5755 results. There are therefore a lot of potential `Customer` components in the database. However, only the first 9 of them involve `Customer` classes, and only one (the 10th) contains the specified methods, but has nothing to do with the notion of customer per se (it is actually a class for analysing HTML code). The only two other relevant classes, which could be used as reference examples, are returned at positions 15 and 21, but both classes called `User` which contains the methods found in the customer class in the running example.

The DAQL dependency-aware version of the search, which accesses the information in the graph database a well as the Lucene index, returns additional customer classes.

```
Defs:{C1:Customer(getName():String;getId(); setName(String))}
```

This query returns 20 results, all of which are `Customer` classes where the required methods are either in the class itself or in a superclass. Table 9.2 show the P@10 and P@5 metrics for the MQL and DAQL version of the search –

MQL	
P@5	P@10
1.0	0.9

DAQL	
P@5	P@10
1.0	1.0

Table 9.4: P@5 and P@10 measurement values for the Customer search with matching methods

With DAQL it is also possible to specify the additional architectural constraint that the `Customer` class has to extends a class which contains a `getId` method –

```
Defs:{C1:Customer(getName():String;setName(String));
      C2:Person(getId())}
Deps:{C1-|>C2}
```

This returns 10 `Customer` classes which all extends a class called `Person`. The first result overrides the `getId` method from the superclass `Person`, whereas the remaining classes in the top 10 simply inherit it (i.e. without overriding). All classes have the following structure.

```
package mv.sub;

public class Customer extends Person{
  private double credit;

  public double getCredit() {
    return credit;
  }

  public void setCredit(double credit) {
    this.credit = credit;
  }
}
```

Listing 9.1: Customer class without the methods

```
package mv.sub;

public class Person {
  private long id;
  private String name;

  public long getId() {
    return id;
  }
  public void setId(long id) {
    this.id = id;
  }
  public String getName() {
    return name;
  }
  public void setName(String name) {
    this.name = name;
  }
}
```

Listing 9.2: Person class containing the desired methods

In the context of a database-driven application, the second result is also directly relevant as it is also annotated with some entity-relation, mapper-specific annotations. Depending on the database used in the application, this class might potentially be reusable without any adaptation. To achieve the same results with Merobase it is necessary to perform two independent searches, one for the `Customer`, but without the `getId` method, and one for the `Person` class. Although relevant results for both classes can be found in the top 10, they both have to be adapted to fit together in the new system.

The P@10 and P@5 metrics for the Merobase searches were obtained by combining the values of the two individual searches. The values P@10 and P@5 values for `Customer` were 0.9 and 1.0, respectively, and for `Person` were 0.8 and 0.6. The poor result for the search for `Person` stems from the fact that the first two results have nothing in common with a `Person` class, but instead merely contain a `getId` method.

MQL	
P@5	P@10
0.85	0.8

DAQL	
P@5	P@10
1.0	1.0

Table 9.5: P@5 and P@10 measurement values of the Customer search

Again a scoring issue arises in the "customer extends person" example because according to the search query, the first `Customer` class which overrides the `getID()` method from the `Person` class is relevant, but to satisfy the user's requirements it must appear later in the list. Nevertheless, all classes found by DAISI can be reused "as-is" without change. Although the database constraint is a requirement of the user in this scenario, it is not actually specified in the search query. If this is a "must have" criteria the query would have to be changed to include the required annotations, but this is currently not supported by DAISI.

Even though this example is again a quite simple use case, it demonstrates that dependency-aware searches, in which it is possible to include the requirement for inheritance into the search query, can significantly boost the quality of the search results in terms of both recall and precision.

9.3 Complex Scenario

The two previous use cases could be supported using Merobase by executing multiple search queries and reformulations. The final search example we discuss in this subsection cannot be achieved using Merobase or any other existing code search engine, so it is not possible to compare P@10 and P@5 values. This example extends the `CustomerManagement` example from chapter 1 with a few additional

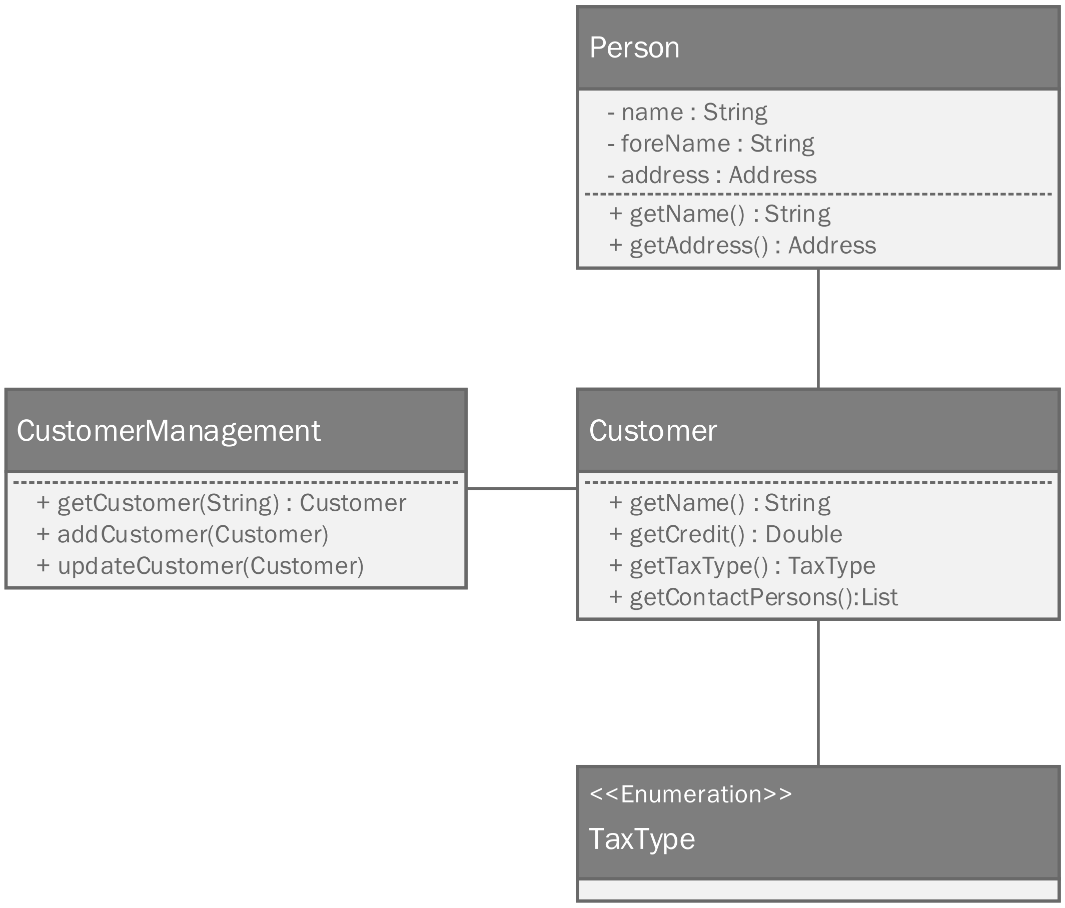


Figure 9.5: Complex search scenario

classes. Figure 9.5 shows that, as in the previous example, the `Customer` is required to extend a `Person` class and every `Customer` is required to additionally possess a relationship to a `TaxType` class. For example, such a tax related class might be necessary in a system to determine if the customer is living in a particular country. Alternatively, a shop might sell items to private and business customers who have different tax classifications.

In traditional code search engines like Merobase a developer would have to search for each one of these classes separately and attempt to reformulate the queries to home in on relevant results. Performing a berrypicking process of the kind described by Bates et al. [Bat89] for each individual class would take a lot of time and have little guarantee of success, so it is highly likely that developers will simply develop the required classes themselves.

DAISI dependency-aware searches can dramatically speed up the process and thus increase the chances that developers will at least give reuse a chance. A DAQL query to support the new use cases would have the form -

```
Defs:{
    C2:CustomerManagement(
       getCustomer():Customer);
   C1:Customer(
      getCredit():Double;
      getTaxType():TaxType);
   C3:TaxType();
   C4:Person(
      getName():String)
} Deps:{C2->C1;C2->C3;C1-|>C4}
```

and delivers the following results:

1. CustomerManagement

   Methods: getCustomer():Customer, addCustomer(Customer), getCustomer-ByTaxType()

2. Customer

   Methods: getName():String, getCredit():Double, getTaxType():TaxType, get-Category()

3. Person

   Methods: getName():String, getAddress():Address, getEmail():String, getId():String

The DAISI search engine returns only one relevant result for this query at the first position. However, the other results are the other classes defined in the query above, like the *Person* or *Customer* class. Nevertheless, this is an extremely successful result since it represents a fully implemented subsystem with all the desired properties. The P@10 and P@5 values for this search are both 1.0. Of course, in contrast to the commercial search engines the index used for this evaluation is relatively small. To gain a better understanding of the effectiveness of the approach it is therefore necessary to try it out on larger repositories. Compared to the search results from chapter 1, where it was quite hard to get appropriate results from other code search engines, DAISI dependency-aware searches provide a highly intuitive way to find relevant results that fits all the extra requirements. Even if queries are quite complex and potentially long, they are written in a format which most developers should find familiar since it is based on the UML notation for method signatures. As explained in the previous chapter, it is also possible to formulate the query in a diagrammatic way, either in the from of a UML diagram or using the "Drag and Drop" search interface.

9.4 Hypothesis Validity

The new dependency-aware search technology described in this thesis and prototyped in DAISI successfully demonstrates the validity of the research hypotheses outlined in chapter 1.

Hypothesis 1

It is possible to build a scalable, language-agnostic, dependency-aware code search engine populated through context-independent harvesting

Result

This has been demonstrated to be valid by the construction of the DAISI prototype search engine using a graph-based database driven by a carefully defined metamodel. This shows that it is possible create and search over dependency-aware, graph-based software data structures. Furthermore, by ensuring that the metamodel uses only the core constructs of object-oriented programming implemented by most languages, the capabilities demonstrated by DAISI are language-agnostic and are thus generally applicable. Finally, by using a multi-phase, context-independent crawling and analysis process DAISI demonstrates that it is possible to infer relationships between classes which may not be explicitly recorded in the environment from which they were harvested.

Threats to Validity

Since the hypothesis is an existential claim (i.e. that a dependency-aware code search engine can be constructed), only one successful implementation is needed to validate the claim. There are therefore no threats to the validity of this conclusion. In particular, since no claim is made about the generalizability of the implementation, there are no external threats to validity.

Hypothesis 2

A dependency-aware code search engine of the kind referred to in Hypothesis 1, which allows users to express the dependency relationships they desire between code elements when defining queries, can enhance the precision of search results.

Result

The evaluation includes several search examples supported by both DAISI and Merobase (through multiple consecutive searches), which showed that the former returned far fewer irrelevant results. For example, in the first scenario presented in chapter 9, Merobase returned the class *NewBookOrderControl* which does not have any relevant methods or structures consistent with a *Stack* of *Items*. Furthermore, because Merobase requires two separate searches to be performed to fulfil the goal, in the second example (section 9.1.2) it provides a lot of irrelevant results since it only considers the method *getName*.

Threats to Validity

A potential internal threat to the validity of any evaluation with such a small set of examples (i.e. sample size) is that the selected examples are not representative of the general population (in this case the set of all possible search queries). However, since the DAISI prototype builds on the Merobase platform, and will always deliver at least as good precision as Merobase, even just one example of improved precision demonstrates an improvement over all. No claim is made about the generalizability of the approach to other platforms, so there is no threat to external validity.

Hypothesis 3

A dependency-aware code search engine of the kind referred to in Hypothesis 1 can enhance the (local) recall of search results.

Result

The evaluation also demonstrates the increase in local recall provided by the new technology, even for traditional searches. Local recall is measured in terms of the components actually stored in the repository. For example, in the simple stack search case, Merobase only returned three relevant results, while DAISI returned six. In fact, DAISI returned every directly relevant result in the index. This is because Merobase requires the results from two separate searches to be manually combined, both of which contain a lot of irrelevant results. On the hand, DAISI automatically combines results that satisfy both requirements, and is able to rank the most relevant results highest. In the stack example, the top three results from DAISI can be reused without any adaptation.

Threats to Validity

A potential internal threat to the validity of any evaluation with such a small set of examples (i.e. sample size) is that the selected examples are not representative of the general population (in this case the set of all possible search queries). However, since the DAISI prototype builds on the Merobase platform, and will always deliver at least as good local recall as Merobase, even just one example of improved recall demonstrates an improvement over all. No claim is made about the generalizability of the approach to other platforms, so there is no threat to external validity.

Hypothesis 4

A dependency-aware code search engine of the kind referred to in Hypothesis 1, populated by a context-independent harvesting approach, can enhance the (global) recall of search results.

Result

In contrast to local recall, global recall takes into account a search engine's ability to harvest components in the first place. DAISI's global recall is increased primarily by its context-independent harvesting capability which allows it to establish relationships that are not contained in the local environment of the analysed code component, but instead are available in the already crawled dataset. DAISI's improved global recall is demonstrated by the second case study (9.1.2 in chapter 9) where the collection of components returned in the first result were not harvested from the same place. For example, the *Stack* class and *Item* class in the first result were harvested from completely different sources, and the former does not even contain a reference to a class called *Item*. DAISI was able to establish the required relationship between the two at search time by virtue of the fact that the *Item* class is a subclass of the class *Object* which the *Stack* class stores.

Threats to Validity

A potential internal threat to the validity of any evaluation with such a small set of examples (i.e. sample size) is that the selected examples are not representative of the general population (in this case the set of all possible search queries). However, since the DAISI context-independent harvesting capability builds on the standard Merobase crawler technology it merely expands the set of components harvests, and thus can only increase recall. Just one example of improved recall therefore demonstrates an improvement over all. No claim is made about the generalizability of the approach to other platforms, so there is no threat to external validity.

10. Conclusion

Computer science is no more about computers than astronomy is about telescopes.

– Edsger Dijkstra –

A major weakness of today's code search engines is that they are only able to support searches for relatively simple components [SCH98]. The principal reason why most developers still primarily use general purpose search engines like Google to search for source code is the lack of support for software structures that extend beyond a single class. This is because they have focused almost exclusively on analysing the contents of individual classes to make them searchable via standard text-processing tools such as Lucene. Although it is potentially possible, with some effort, to use Lucene to support searches for groups of related classes, this involves a process of repeated query reformulation within a series of sequential sub searches. As well as being inefficient, such multiple-consecutive-search approaches also yield questionable results. This particularly relates to relationships involving methods used from other classes, because current search engines store only method signatures and the relationships between method names and functionality is a tenuous one.

The main research trend over the last few years has been in the direction of code recommendation where statistical analysis techniques are used to suggest code fragments and method call sequences to developers based on the current state of their code. However, the underlying approaches used to discover the candidate code fragments and components in the first place are still driven by traditional search technologies.

The aim of this thesis was therefore to create the foundations for a new generation of code search engines capable of supporting more sophisticated searches that take the structure of complex software components into account - that is, are "aware of", and can exploit, the relationships between inter-related classes. Moreover, these foundations should be independent of specific programming languages and relationships so they can be adapted to support new programming languages and features in the future (e.g. such as in Java 8 introduced lambda expressions). To realize this goal, a new kind of database structure based on graphs was developed to store software components and a new, dependency-aware query language was developed to accommodate constraints on relationships between classes and methods. Finally, to populate this new kind of database with content, a new parsing process was developed capable of analysing individual classes in a context free manner without requiring information from complete projects to identify the relationships between classes. This parsing technology is a key ingredient, since a search engine should be able to harvest all the code available in the Internet, not just the classes wrapped up in complete projects controlled by configuration management systems.

To develop an optimal database structure, the Lucene text-based indexing system that underlies the majority of modern code search engines was integrated in a sophisticated way with a fundamentally new kind of database – a graph database. This combination was considered optimal because, while the graph database provides the ideal way of storing the many kinds of links between classes, Lucene provides the optimal way of discovering initial sets of candidates, and thus of defining starting points for graph-driven searches. In contrast to existing search engines, to combine them in an effective way, several Lucene indices were used to store different kinds of information about the harvested components in the most efficient way. To increase the range of search queries that could be supported, the MQL query language defined by Hummel for the Merobase search engine was extended with dependency-awareness. This extension makes it possible to enhance traditional MQL searches with constraints about the relationships that should exist between the desired classes. Although the DAQL query language is as simple and intuitive as possible, dependency aware searches become unavoidably complex when expressed in textual format. For this reason additional, non-text based ways of expressing dependency-aware queries were also developed – namely, diagrammatic query languages based on UML class diagrams and a web-based "Drag and Drop" query definition interface.

In combination, these innovations introduce a range of new search capabilities to users of code search engines. More specifically, they make it possible to –

- search for classes that possess certain structural relationships,
- search for multiple components with only one query,
- harvest code outside the context of a complete project structure,
- define dependency-aware queries without learning a complex, text-based query language.

10.1 Weaknesses

Although the example dependency-aware searches used in the evaluation demonstrate significant increases in recall and precision as measured by the P@10 and P@5 metrics, there are some weaknesses in the current prototype implementation. The first weakness, which was also visible with the simplest stack example, is the way that the relevance of search results are evaluated for ranking. To address this problem a new evaluation mechanism needs to be integrated which is ideally performed after the graph database has been seed to filter out unsuitable components.

Another weakness is the complexity of the query language. The significance of query language complexity is highlighted by various studies which show that users prefer to provide as little information as possible in search queries and rarely use the special prefixes offered by existing search engines. However, to obtain the kind of precision and recall provided by dependency-aware search engines like DAISI it is necessary for users to indicate which keywords are expected to fulfil what role in candidate components. Rather than reduce the information that has to be supplied in queries, in the previous chapter we presented approaches that allow users to supply this information in an as intuitive and simple a way as possible - name, via diagrammatic and "Drag and Drop" based queries.

The final weakness relates to the strictness of the matching criteria in dependency-aware searches. The more complex a search query, the more constraints have to be matched by candidates in the search repository. If all constraints are applied strictly in an all-or-nothing fashion, many components that could be potentially useful for the user may not be returned because they deviate from the requirements in a small way. For instance, consider the customer example. If no class can be found which satisfies all the constraints specified in the query, the user might still be able to exploit the "*Person*"-class result as a reference example. However, this would not be returned in a strict interpretation of the constraints since all classes would be removed from the result list. This leads to a kind of reverse scenario where the user no longer has to extend his/her search query with additional information step-by-step, but instead has to remove constraints from the query step by step. Moreover, while doing this he/she would have to be aware of the different combinations in which the constraints could be satisfied. For example, removal of a return parameter constraint on a method might not deliver any new results whereas the removal of a complete method might.

10.2 Future Work

DAISI's ability to ensure all required dependencies between classes are included in search queries and to examine the structure of components and projects opens up a lot of new opportunities. This new technology can not only be used to enhance the capabilities of existing code search engines, but can also lead to completely new opportunities. First of all, it is not only possible to consider dependencies directly-expressed in the source code, it is also possible to consider indirectly derived dependencies. For example, the dependencies derived from a social-media inspired dislike/like mechanism can be exploited as discussed in chapter 6 to add different metrics to the relationships between the potential candidates in a dependency. These metrics could be performance metrics or energy efficient metrics for example. Energy efficiency metrics could be particularly interesting, as many developers in the mobile sector are currently developing applications that have power consumption constraints.

Another possibility is to create different enhancements of comparative metrics or analysis approaches. Currently only two classes are compared to each other to generate such metrics, but what if a copy is created in which the original class is divided into two classes by moving some of the functionality into a superclass? No existing algorithm today would recognize such a class as a clone of the original.

Another opportunity created by the new technology is related to the open source sector. As Spinelli and Szyperski have already pointed out the quality of software in the open source sector is subject to large fluctuations [SS04]. There are many shoddy components of very low quality and there are many high-quality components that meet industrial standards. Since the quality of new systems is affected by the quality of the components they are made of, one useful extension of the search engine would be to keep track of different versions of components as they are tested and improved. Thus, when a developer is upgrading a system he/she could include an improved component, whilst keeping the relationship to the original one. This relationship could then be characterized by metrics or other properties. The next developer who recommended this component at the search engine would be informed automatically about new, improved versions of the component. Of course, over time a mechanism would be needed to fix discovered bugs in all existing versions, or periodically these different versions would have to be merged.

Bibliography

Books

[B+99] Ricardo Baeza-Yates, Berthier Ribeiro-Neto, et al. *Modern information retrieval*. Volume 463. ACM press New York, 1999 (cited on pages 21, 23, 57).

[Car85] Alfonso F Cardenas. *Data base management systems*. Volume 2. Allyn and Bacon Boston et al., 1985 (cited on page 43).

[CMS10] W Bruce Croft, Donald Metzler, and Trevor Strohman. *Search engines: Information retrieval in practice*. Volume 283. Addison-Wesley Reading, 2010 (cited on pages 20, 25, 36, 37, 113).

[GS14] Rosalva E. Gallardo-Valencia and Susan Elliott Sim. *Source Code Seeking on the Web: A Survey of Empirical Studies and Tools*. Lulu.com, 2014. ISBN: 130469545X, 9781304695451 (cited on page 66).

[HJD10] Elizabeth Hull, Ken Jackson, and Jeremy Dick. *Requirements engineering*. Springer Science & Business Media, 2010 (cited on page 55).

[Jan14] W. Janjic. *Reuse-Based Test Recommendation in Software Engineering*. Verlag Dr. Hut, 2014. ISBN: 9783843916738. URL: https://books.google.com/books?id=wU7HoQEACAAJ (cited on pages 3, 7).

[Kru04] Philippe Kruchten. *The rational unified process: an introduction*. Addison-Wesley Professional, 2004 (cited on page 68).

[Lov68] Julie B Lovins. *Development of a stemming algorithm*. MIT Information Processing Group, Electronic Systems Laboratory Cambridge, 1968 (cited on page 38).

[MHG10] Michael McCandless, Erik Hatcher, and Otis Gospodnetic. *Lucene in Action: Covers Apache Lucene 3.0*. Manning Publications Co., 2010 (cited on pages 40, 49, 50).

[Min69] Marvin L. Minsky. *Semantic Information Processing*. The MIT Press, 1969. ISBN: 0262130440 (cited on page 27).

[Mit98] Roland T Mittermeir. *Hypertext: Werkzeug?–Denkzeug?* na, 1998 (cited on page 32).

[OMG11a] OMG. *Architecture-driven Modernization: Abstract Syntax Tree Metamodel (ASTM)- Version 1.0*. Object Management Group, Jan. 2011. URL: http://www.omg.org/spec/ASTM/1.0/PDF/ (cited on pages 94, 97).

[OMG11b] OMG. *Architecture-Driven Modernization: Knowledge Discovery Meta-Model (KDM) - Version 1.3*. Object Management Group, Aug. 2011. URL: http://www.omg.org/spec/KDM/1.3/PDF/ (cited on page 97).

[Par13] Terence Parr. *The Definitive ANTLR 4 Reference*. 2nd. Pragmatic Bookshelf, 2013. ISBN: 1934356999, 9781934356999 (cited on page 94).

[Sal68] Gerard. Salton. *Automatic Information Organization and Retrieval.* McGraw Hill Text, 1968. ISBN: 0070544859 (cited on page 19).

[SM83] Gerard Salton and Michael J McGill. *Introduction to modern information retrieval*. New York: McGraw - Hill Book Company, 1983. ISBN: 0070544840 (cited on page 29).

[Som01] I. Sommerville. *Software engineering*. International computer science series. Addison-Wesley, 2001. ISBN: 9780201398151. URL: https://books.google.co.uk/books?id=Uoo%5C_AQAAIAAJ (cited on pages 1, 18, 126, 127, 129).

[Szy02] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. 2nd. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN: 0201745720 (cited on page 68).

[Vuk+15] Aleksa Vukotic et al. *Neo4j in Action*. Manning, 2015 (cited on page 43).

[Atk+08] Colin Atkinson et al. “Modeling Components and Component-Based Systems in KobrA”. In: *The Common Component Modeling Example: Comparing Software Component Models*. Edited by Andreas Rausch et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pages 54–84. ISBN: 978-3-540-85289-6. DOI: 10.1007/978-3-540-85289-6_4. URL: https://doi.org/10.1007/978-3-540-85289-6_4 (cited on page 137).

Articles

[All+88] Thomas Allen et al. “Project team aging and performance: The roles of project and functional managers”. In: *R&D Management* 18.4 (1988), pages 295–308 (cited on page 129).

[BL12a] Sushil Krishna Bajracharya and Cristina Videira Lopes. “Analyzing and mining a code search engine usage log”. In: *Empirical Software Engineering* 17.4-5 (2012), pages 424–466 (cited on pages 62, 104, 113, 114, 119, 126).

[BOL14] Sushil Bajracharya, Joel Ossher, and Cristina Lopes. “Sourcerer: An infrastructure for large-scale collection and analysis of open-source code”. In: *Science of Computer Programming* 79 (2014), pages 241–259 (cited on pages 40, 41, 58).

[Bat89] Marcia J Bates. “The design of browsing and berrypicking techniques for the online search interface”. In: *Online Review* 13, No. 5 (1989), pages 407–424 (cited on pages 148, 161).

[BOB82] Nicholas J Belkin, Robert N Oddy, and Helen M Brooks. “ASK for information retrieval: Part I. Background and theory”. In: *Journal of documentation* 38.2 (1982), pages 61–71 (cited on pages 37, 59).

[BMW94] Ted J Biggerstaff, Bharat G Mitbander, and Dallas E Webster. “Program understanding and the concept assignment problem”. In: *Communications of the ACM* 37.5 (1994), pages 72–82 (cited on pages 31, 55).

[BL12b] Roi Blanco and Christina Lioma. “Graph-based Term Weighting for Information Retrieval”. In: *Inf. Retr.* 15.1 (Feb. 2012), pages 54–92. ISSN: 1386-4564. DOI: 10.1007/s10791-011-9172-x. URL: http://dx.doi.org/10.1007/s10791-011-9172-x (cited on pages 26–28).

[BF93] Nathaniel Borenstein and Ned Freed. “MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for specifying and describing the format of Internet message bodies”. In: (1993) (cited on page 90).

[BP12] Sergey Brin and Lawrence Page. “Reprint of: The anatomy of a large-scale hypertextual web search engine”. In: *Computer networks* 56.18 (2012), pages 3825–3833 (cited on page 89).

[Cho+02] Abdur Chowdhury et al. “Collection statistics for fast duplicate document detection”. In: *ACM Transactions on Information Systems (TOIS)* 20.2 (2002), pages 171–191 (cited on page 77).

[Cle61] Cyril W Cleverdon. “Application for grant to the National Science Foundation, Washington [for] an investigation into the performance characteristics of descriptor languages”. In: (1961) (cited on pages 21, 39).

[Cus89] Michael A Cusumano. “The software factory: a historical interpretation”. In: *IEEE Software* 6.2 (1989), page 23 (cited on page 19).

[Dee+90] Scott Deerwester et al. “Indexing by latent semantic analysis”. In: *Journal of the American society for information science* 41.6 (1990), page 391 (cited on pages 38, 53).

[FCC07] Ramon Ferrer I Cancho, Andrea Capocci, and Guido Caldarelli. “Spectral Methods Cluster Words of the Same Class in a Syntactic Dependency Network”. In: *Int. J. Bifurc. Chaos* 17.07 (2007), pages 2453–2462. ISSN: 0218-1274. DOI: 10.1142/S021812740701852X. URL: http://www.worldscinet.com/ijbc/17/1707/S021812740701852X.html (cited on page 26).

[FB92] William B Frakes and Ricardo Baeza-Yates. “Information retrieval: data structures and algorithms”. In: (1992) (cited on page 29).

[FP94] William B. Frakes and Thomas P. Pole. “An empirical study of representation methods for reusable software components”. In: *IEEE Transactions on Software Engineering* 20.8 (1994), pages 617–630 (cited on page 34).

[Fur+87] George W. Furnas et al. “The vocabulary problem in human-system communication”. In: *Communications of the ACM* 30.11 (1987), pages 964–971 (cited on pages 30, 53, 56).

[GP99] Michael Gordon and Praveen Pathak. “Finding information on the World Wide Web: the retrieval effectiveness of search engines”. In: *Information Processing & Management* 35.2 (1999), pages 141–180 (cited on page 21).

[HH76] Michael AK Halliday and Ruqaiya Hasan. “Cohesion in English”. In: (1976) (cited on page 26).

[HDK06] Björn Hartmann, Scott Doorley, and Scott R Klemmer. “Hacking, mashing, gluing: a study of opportunistic design and development”. In: *Pervasive Computing* 7.3 (2006), pages 46–54 (cited on page 126).

[Har+03] Nicholas JA Harvey et al. “Skipnet: A scalable overlay network with practical locality properties”. In: *networks* 34 (2003), page 38 (cited on page 43).

[Kru92] Charles W. Krueger. “Software Reuse”. In: *ACM Comput. Surv.* 24.2 (June 1992), pages 131–183. ISSN: 0360-0300. DOI: 10.1145/130844.130856. URL: http://doi.acm.org/10.1145/130844.130856 (cited on pages 17, 48).

[Kuk92] Karen Kukich. “Techniques for automatically correcting words in text”. In: *ACM Computing Surveys (CSUR)* 24.4 (1992), pages 377–439 (cited on page 38).

[LM89] B. M. Lange and T. G. Moher. “Some Strategies of Reuse in an Object-oriented Programming Environment”. In: *SIGCHI Bull.* 20.SI (Mar. 1989), pages 69–73. ISSN: 0736-6906. DOI: 10.1145/67450.67465. URL: http://doi.acm.org/10.1145/67450.67465 (cited on page 1).

[Lim94] W.C. Lim. “Effects of reuse on quality, productivity, and economics”. In: *Software, IEEE* 11.5 (Sept. 1994), pages 23–30. ISSN: 0740-7459. DOI: 10.1109/52.311048 (cited on page 2).

[Lin+09] Erik Linstead et al. “Sourcerer: mining and searching internet-scale software repositories”. In: *Data Mining and Knowledge Discovery* 18.2 (2009), pages 300–336 (cited on pages 57, 58).

[MBK91] Yoëlle S Maarek, Daniel M Berry, and Gail E Kaiser. “An information retrieval approach for automatically constructing software libraries”. In: *IEEE Transactions on software Engineering* 17.8 (1991), pages 800–813 (cited on pages 30, 34).

[MK14] Anita Brigit Mathew and SM Kumar. “An efficient index based query handling model for Neo4j”. In: *IJCST* 3.2 (2014), pages 12–18 (cited on page 43).

[Mat84] Yoshihiro Matsumoto. “Some experiences in promoting reusable software: Presentation in higher abstract levels”. In: *IEEE Transactions on Software Engineering* 5 (1984), pages 502–513 (cited on page 19).

[McI68] M. D. McIlroy. “Mass-produced software components”. In: *Proc. NATO Conf. on Software Engineering, Garmisch, Germany* (1968) (cited on page 1).

[MMM98] A. Mili, R. Mili, and R. T. Mittermeir. “A Survey of Software Reuse Libraries”. In: *Ann. Softw. Eng.* 5.1 (Jan. 1998), pages 349–414. ISSN: 1022-7091. URL: http://dl.acm.org/citation.cfm?id=590631.590637 (cited on pages 30, 31, 34, 35).

[Mil+99] Ali Mili et al. “Toward an engineering discipline of software reuse”. In: *IEEE software* 16.5 (1999), pages 22–31 (cited on page 38).

[Mil+90] George A Miller et al. “Introduction to WordNet: An on-line lexical database”. In: *International journal of lexicography* 3.4 (1990), pages 235–244 (cited on page 53).

[Moo50] Calvin E. Mooers. “Coding, Information Retrieval, and the Rapid Selector”. In: *American Documentation* 1.4 (1950), pages 225–229 (cited on page 19).

[MMD02] A. E. Motter, A. P. S. de Moura, and P. Dasgupta. “Topology of the conceptual network of language”. In: *Physical Review E* 65.065102(R) (2002) (cited on page 26).

[MKF06] Gail C Murphy, Mik Kersten, and Leah Findlater. “How are Java software developers using the Elipse IDE?” In: *IEEE software* 23.4 (2006), pages 76–83 (cited on page 18).

[MNS95] Gail C Murphy, David Notkin, and Kevin Sullivan. “Software reflexion models: Bridging the gap between source and high-level models”. In: *ACM SIGSOFT Software Engineering Notes* 20.4 (1995), pages 18–28 (cited on page 55).

[Ort+16] Fernando Ortega et al. “Recommending Items to Group of Users Using Matrix Factorization Based Collaborative Filtering”. In: *Inf. Sci.* 345.C (June 2016), pages 313–324. ISSN: 0020-0255. DOI: `10.1016/j.ins.2016.01.083`. URL: `https://doi.org/10.1016/j.ins.2016.01.083` (cited on page 143).

[PP93] Andy Podgurski and Lynn Pierce. “Retrieving reusable software by sampling behavior”. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 2.3 (1993), pages 286–303 (cited on page 32).

[SFW83] Gerard Salton, Edward A. Fox, and Harry Wu. “Extended Boolean Information Retrieval”. In: *Commun. ACM* 26.11 (Nov. 1983), pages 1022–

1036. ISSN: 0001-0782. DOI: 10.1145/182.358466. URL: http://doi.acm.org/10.1145/182.358466 (cited on page 23).

[Sim+11] Susan Elliott Sim et al. "How Well Do Search Engines Support Code Retrieval on the Web?" In: *ACM Trans. Softw. Eng. Methodol.* 21.1 (Dec. 2011), 4:1–4:25. ISSN: 1049-331X. DOI: 10.1145/2063239.2063243. URL: http://doi.acm.org/10.1145/2063239.2063243 (cited on pages 2–4, 126).

[SS04] Diomidis Spinellis and Clemens Szyperski. "Guest Editors' Introduction: How Is Open Source Affecting Software Development?" In: *IEEE Softw.* 21.1 (Jan. 2004), pages 28–33. ISSN: 0740-7459. DOI: 10.1109/MS.2004.1259204. URL: http://dx.doi.org/10.1109/MS.2004.1259204 (cited on page 173).

[SJC00] Amanda Spink, Bernard J Jansen, and H Cenk Ozmultu. "Use of query reformulation and relevance feedback by Excite users". In: *Internet research* 10.4 (2000), pages 317–328 (cited on page 156).

[SED14] Kathryn T. Stolee, Sebastian Elbaum, and Daniel Dobos. "Solving the Search for Source Code". In: *ACM Trans. Softw. Eng. Methodol.* 23.3 (June 2014), 26:1–26:45. ISSN: 1049-331X. DOI: 10.1145/2581377. URL: http://doi.acm.org/10.1145/2581377 (cited on page 131).

[Woo+96] Allison Woodruff et al. "An investigation of documents from the World Wide Web". In: *Computer Networks and ISDN Systems* 28.7 (1996), pages 963–980 (cited on page 90).

[ZW95] Amy Moormann Zaremski and Jeannette M Wing. "Signature matching: a tool for using software libraries". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 4.2 (1995), pages 146–170 (cited on pages 31, 48).

[AIS93] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. "Mining association rules between sets of items in large databases". In: *Acm sigmod record*. Volume 22. 2. ACM. 1993, pages 207–216 (cited on page 25).

[AG15] Carol V. Alexandru and Harald C. Gall. "Rapid Multi-purpose, Multi-commit Code Analysis". In: *Proceedings of the 37th International Conference on Software Engineering - Volume 2*. ICSE '15. Florence, Italy: IEEE Press, 2015, pages 635–638. URL: `http://dl.acm.org/citation.cfm?id=2819009.2819124` (cited on page 34).

[Atk97] Colin Atkinson. "Meta-modelling for distributed object environments". In: *Enterprise Distributed Object Computing Workshop [1997]. EDOC'97. Proceedings. First International*. IEEE. 1997, pages 90–101 (cited on page 98).

[Atk+09] Colin Atkinson et al. "Towards high integrity uddi systems". In: *International Conference on Business Information Systems*. Springer. 2009, pages 350–361 (cited on page 111).

[Baj+06] Sushil Bajracharya et al. "Sourcerer: a search engine for open source code supporting structure-based search". In: *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. ACM. 2006, pages 681–682 (cited on pages 34, 57).

[Beg07] Andrew Begel. "Codifier: a programmer-centric search user interface". In: *Proceedings of the workshop on human-computer interaction and information retrieval*. 2007, pages 23–24 (cited on page 34).

[BHQ03] Stefan Bordag, Gerhard Heyer, and Uwe Quasthoff. "Small Worlds of Concepts and Other Principles of Semantic Search". In: *Innovative Internet Community Systems, Third International Workshop, IICS 2003, Leipzig, Germany, June 19-21, 2003, Revised Papers*. 2003, pages 10–

19. DOI: 10.1007/978-3-540-39884-4_2. URL: http://dx.doi.org/10.1007/978-3-540-39884-4_2 (cited on page 26).

[Bra+09] Joel Brandt et al. “Two studies of opportunistic programming: interleaving web foraging, learning, and writing code”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM. 2009, pages 1589–1598 (cited on page 125).

[CJS09] Shaunak Chatterjee, Sudeep Juvekar, and Koushik Sen. “Sniff: A search engine for java using free-form queries”. In: *International Conference on Fundamental Approaches to Software Engineering*. Springer. 2009, pages 385–400 (cited on page 56).

[Cho+07] M. Choudhury et al. “How Difficult is it to Develop a Perfect Spell-checker? A Cross-linguistic Analysis through Complex Network Approach”. In: *Proceedings of the Second Workshop on TextGraphs: Graph-Based Algorithms for Natural Language Processing*. 2007, pages 81–88 (cited on page 26).

[Cle91] Cyril W. Cleverdon. “The Significance of the Cranfield Tests on Index Languages”. In: *Proceedings of the 14th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR ’91. Chicago, Illinois, USA: ACM, 1991, pages 3–12. ISBN: 0-89791-448-1. DOI: 10.1145/122860.122861. URL: http://doi.acm.org/10.1145/122860.122861 (cited on page 19).

[GCP07] Mark Grechanik, Kevin M Conroy, and Katharina A Probst. “Finding relevant applications for prototyping”. In: *Proceedings of the Fourth International Workshop on Mining Software Repositories*. IEEE Computer Society. 2007, page 12 (cited on page 56).

[Gre+10a] Mark Grechanik et al. “A search engine for finding highly relevant applications”. In: *2010 ACM/IEEE 32nd International Conference on*

Software Engineering. Volume 1. IEEE. 2010, pages 475–484 (cited on page 56).

[Gre+10b] Mark Grechanik et al. “Exemplar: Executable examples archive”. In: *2010 ACM/IEEE 32nd International Conference on Software Engineering*. Volume 2. IEEE. 2010, pages 259–262 (cited on pages 31, 48, 55, 56).

[GJ09] Pooja Gupta and Kalpana Johari. “Implementation of Web crawler”. In: *2009 Second International Conference on Emerging Trends in Engineering & Technology*. IEEE. 2009, pages 838–843 (cited on page 90).

[Hai+13] Sonia Haiduc et al. “Automatic query reformulations for text retrieval in software engineering”. In: *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press. 2013, pages 842–851 (cited on page 36).

[HFW07] Raphael Hoffmann, James Fogarty, and Daniel S Weld. “Assieme: finding and leveraging implicit references in a web search interface for programmers”. In: *Proceedings of the 20th annual ACM symposium on User interface software and technology*. ACM. 2007, pages 13–22 (cited on page 34).

[HM05] Reid Holmes and Gail C. Murphy. “Using Structural Context to Recommend Source Code Examples”. In: *Proceedings of the 27th International Conference on Software Engineering*. ICSE ’05. St. Louis, MO, USA: ACM, 2005, pages 117–125. ISBN: 1-58113-963-2. DOI: 10.1145/1062455.1062491. URL: http://doi.acm.org/10.1145/1062455.1062491 (cited on page 3).

[HW07] Reid Holmes and Robert J. Walker. “Supporting the Investigation and Planning of Pragmatic Reuse Tasks”. In: *Proceedings of the 29th*

International Conference on Software Engineering. ICSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pages 447–457. ISBN: 0-7695-2828-7. DOI: 10.1109/ICSE.2007.83. URL: http://dx.doi.org/10.1109/ICSE.2007.83 (cited on pages 1, 126).

[HW08] Reid Holmes and Robert J Walker. "Lightweight, semi-automated enactment of pragmatic-reuse plans". In: *International Conference on Software Reuse*. Springer. 2008, pages 330–342 (cited on page 126).

[HWM05] Reid Holmes, Robert J Walker, and Gail C Murphy. "Strathcona example recommendation tool". In: *ACM SIGSOFT Software Engineering Notes*. Volume 30. 5. ACM. 2005, pages 237–240 (cited on page 63).

[HR92] Susan Horwitz and Thomas Reps. "The use of program dependence graphs in software engineering". In: *Proceedings of the 14th international conference on Software engineering*. ACM. 1992, pages 392–411 (cited on page 44).

[Hum08] Oliver Hummel. "Semantic Component Retrieval in Software Engineering". In: *Ausgezeichnete Informatikdissertationen 2008*. 2008, pages 151–160 (cited on pages 48, 50, 53, 86, 114, 120, 121).

[HA04] Oliver Hummel and Colin Atkinson. "Extreme harvesting: Test driven discovery and reuse of software components". In: *Information Reuse and Integration, 2004. IRI 2004. Proceedings of the 2004 IEEE International Conference on*. IEEE. 2004, pages 66–72 (cited on pages 33, 110).

[HA07] Oliver Hummel and Colin Atkinson. "Supporting agile reuse through extreme harvesting". In: *International Conference on Extreme Programming and Agile Processes in Software Engineering*. Springer. 2007, pages 28–37 (cited on page 74).

[HJA07] Oliver Hummel, Werner Janjic, and Colin Atkinson. "Evaluating the Efficiency of Retrieval Methods for Component Repositories." In: *SEKE*. Citeseer. 2007, pages 404–409 (cited on page 31).

[Jan+13] Werner Janjic et al. "An Unabridged Source Code Dataset for Research in Software Reuse". In: *Proceedings of the 10th Working Conference on Mining Software Repositories*. MSR '13. San Francisco, CA, USA: IEEE Press, 2013, pages 339–342. ISBN: 978-1-4673-2936-1. URL: http://dl.acm.org/citation.cfm?id=2487085.2487148 (cited on pages 3, 48).

[JZW09] Yinan Jing, Chunwang Zhang, and Xueping Wang. "An empirical study on performance comparison of lucene and relational database". In: *Communication Software and Networks, 2009. ICCSN'09. International Conference on*. IEEE. 2009, pages 336–340 (cited on page 83).

[Mar+04] Andrian Marcus et al. "An information retrieval approach to concept location in source code". In: *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*. IEEE. 2004, pages 214–223 (cited on page 31).

[MLL05] Michael Martin, Benjamin Livshits, and Monica S Lam. "Finding application errors and security flaws using PQL: a program query language". In: *ACM SIGPLAN Notices*. Volume 40. 10. ACM. 2005, pages 365–383 (cited on page 37).

[McM+11] Collin McMillan et al. "Portfolio: A Search Engine for Finding Functions and Their Usages". In: *Proceedings of the 33rd International Conference on Software Engineering*. ICSE '11. Waikiki, Honolulu, HI, USA: ACM, 2011, pages 1043–1045. ISBN: 978-1-4503-0445-0. DOI: 10.1145/1985793.1985991. URL: http://doi.acm.org/10.1145/1985793.1985991 (cited on pages 3, 7, 34, 35, 48, 53, 97, 106).

[Moh+04] Parastoo Mohagheghi et al. "An Empirical Study of Software Reuse vs. Defect-Density and Stability". In: *Proceedings of the 26th International Conference on Software Engineering*. ICSE '04. Washington, DC, USA: IEEE Computer Society, 2004, pages 282–292. ISBN: 0-7695-2163-0. URL: http://dl.acm.org/citation.cfm?id=998675.999433 (cited on page 1).

[Ngu+12] Anh Tuan Nguyen et al. "Graph-based Pattern-oriented, Context-sensitive Source Code Completion". In: *Proceedings of the 34th International Conference on Software Engineering*. ICSE '12. Zurich, Switzerland: IEEE Press, 2012, pages 69–79. ISBN: 978-1-4673-1067-3. URL: http://dl.acm.org/citation.cfm?id=2337223.2337232 (cited on page 35).

[OH92] Paul Oman and Jack Hagemeister. "Metrics for assessing a software system's maintainability". In: *Software Maintenance, 1992. Proceerdings., Conference on*. IEEE. 1992, pages 337–344 (cited on page 57).

[Pag+98] L. Page et al. "The PageRank citation ranking: Bringing order to the Web". In: *Proceedings of the 7th International World Wide Web Conference*. Brisbane, Australia, 1998, pages 161–172. URL: citeseer.nj.nec.com/page98pagerank.html (cited on page 27).

[PPZ11] Oleksandr Panchenko, Hasso Plattner, and Alexander Zeier. "What do developers search for in source code and why". In: *Proceedings of the 3rd International Workshop on Search-Driven Development: Users, Infrastructure, Tools, and Evaluation*. ACM. 2011, pages 33–36 (cited on pages 104, 113, 126).

[Pôs+02] Bruno Pôssas et al. "Set-based Model: A New Approach for Information Retrieval". In: *Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information*

Retrieval. SIGIR '02. Tampere, Finland: ACM, 2002, pages 230–237. ISBN: 1-58113-561-0. DOI: 10.1145/564376.564417. URL: http://doi.acm.org/10.1145/564376.564417 (cited on pages 25, 53).

[QLS13] DH Qiu, H Li, and JL Sun. "Measuring software similarity based on structure and property of class diagram". In: *Advanced Computational Intelligence (ICACI), 2013 Sixth International Conference on*. IEEE. 2013, pages 75–80 (cited on page 34).

[Sch+08] Ralf Schenkel et al. "Efficient Top-k Querying over Social-tagging Networks". In: *Proceedings of the 31st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR '08. Singapore, Singapore: ACM, 2008, pages 523–530. ISBN: 978-1-60558-164-4. DOI: 10.1145/1390334.1390424. URL: http://doi.acm.org/10.1145/1390334.1390424 (cited on page 27).

[SA15] Marcus Schumacher and Colin Atkinson. "An Enhanced Graph-based Infrastructure for Software Search Engines". In: *12th Working Conference on Mining Software Repositories, Florence*. 2015 (cited on page 114).

[SCH98] Susan Elliott Sim, Charles L. A. Clarke, and Richard C. Holt. "Archetypal Source Code Searches: A Survey of Software Developers and Maintainers." In: *IWPC*. IEEE Computer Society, 1998, pages 180–187. ISBN: 0-8186-8560-3. URL: http://dblp.uni-trier.de/db/conf/iwpc/iwpc1998.html#SimCH98 (cited on pages 38, 129, 169).

[Sin+97] Janice Singer et al. "An Examination of Software Engineering Work Practices". In: *Proceedings of the 1997 Conference of the Centre for Advanced Studies on Collaborative Research*. CASCON '97. Toronto, Ontario, Canada: IBM Press, 1997, pages 21–. URL: http://dl.acm.org/citation.cfm?id=782010.782031 (cited on page 18).

[Sin+09] Sitabhra Sinha et al. “Network Analysis Reveals Structure Indicative of Syntax in the Corpus of Undeciphered Indus Civilization Inscriptions”. In: *Proceedings of the 2009 Workshop on Graph-based Methods for Natural Language Processing*. TextGraphs-4. Suntec, Singapore: Association for Computational Linguistics, 2009, pages 5–13. ISBN: 978-1-932432-54-1. URL: `http://dl.acm.org/citation.cfm?id=1708124.1708128` (cited on page 27).

[SH13a] Siddharth Subramanian and Reid Holmes. “Making sense of online code snippets”. In: *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR ’13, San Francisco, CA, USA, May 18-19, 2013*. 2013, pages 85–88. DOI: `10.1109/MSR.2013.6624012`. URL: `http://dx.doi.org/10.1109/MSR.2013.6624012` (cited on page 8).

[SH13b] Siddharth Subramanian and Reid Holmes. “Making sense of online code snippets”. In: *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press. 2013, pages 85–88 (cited on pages 67, 81).

[TX07] Suresh Thummalapenta and Tao Xie. “Parseweb: A Programmer Assistant for Reusing Open Source Code on the Web”. In: *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*. ASE ’07. Atlanta, Georgia, USA: ACM, 2007, pages 204–213. ISBN: 978-1-59593-882-4. DOI: `10.1145/1321631.1321663`. URL: `http://doi.acm.org/10.1145/1321631.1321663` (cited on page 35).

[USL08] Medha Umarji, Susan Elliott Sim, and Crista Lopes. “Archetypal internet-scale source code searching”. In: *IFIP International Con-*

ference on Open Source Systems. Springer. 2008, pages 257–263 (cited on page 130).

[WLJ11] Shaowei Wang, David Lo, and Lingxiao Jiang. “Code search via topic-enriched dependence graph matching”. In: *2011 18th Working Conference on Reverse Engineering*. IEEE. 2011, pages 119–123 (cited on pages 37, 44, 97).

[WH91] Ross Wilkinson and Philip Hingston. “Using the cosine measure in a neural network for document retrieval”. In: *Proceedings of the 14th annual international ACM SIGIR conference on Research and development in information retrieval*. ACM. 1991, pages 202–210 (cited on page 27).

[Hop88] J. J. Hopfield. “Neurocomputing: Foundations of Research”. In: edited by James A. Anderson and Edward Rosenfeld. Cambridge, MA, USA: MIT Press, 1988. Chapter Neural Networks and Physical Systems with Emergent Collective Computational Abilities, pages 457–464. ISBN: 0-262-01097-6. URL: `http://dl.acm.org/citation.cfm?id=65669.104422` (cited on page 27).

[HAS13] Oliver Hummel, Colin Atkinson, and Marcus Schumacher. “Artifact representation techniques for large-scale software search engines”. In: *Finding Source Code on the Web for Remix and Reuse*. Springer, 2013, pages 81–101 (cited on pages 10, 41, 42, 71).

[Kru13] Ken Krugler. “Krugle code search architecture”. In: *Finding Source Code on the Web for Remix and Reuse*. Springer, 2013, pages 103–120 (cited on pages 2, 59–61, 91).

[Lid05] Elizabeth D. Liddy. “Automatic Document Retrieval”. In: *Encyclopedia of Language and Linguistics*. 2nd. Elsevier, 2005 (cited on page 19).