Technische Universität München
Fakultät für Informatik

Supporting Reuse in Evolving Code Bases using Code Search

Thomas Kinnen

Master's Thesis in Computer Science

Technische Universität München
Fakultät für Informatik

Supporting Reuse in Evolving Code Bases using Code Search

Suchen von wiederverwendbarem Code in evolutionär entwickelter Software

Thomas Kinnen

Master's Thesis in Computer Science

Supervisor: Prof. Dr. Dr. h.c. Manfred Broy

Advisors: M.Sc. Veronika Bauer
Dr. Elmar Jürgens
Dr. Lars Heinemann

Submission Date: 15. October 2013

Declaration

I assure the single handed composition of this master's thesis only supported by declared resources.

Munich, 15. October 2013

.....
Thomas Kinnen

Abstract

Research in software engineering has shown that the reuse of software components reduces bugs, improves code quality and decreases development time.

In the past many code search systems have been proposed to help developers find code that is suitable for reuse in large code bases. Today large companies see several commits to their software repositories every minute and developers expect information they rely on to be up to date. This creates the need for a novel approach of analyzing the code base and keeping the code search engine updated. The traditional approach to read the entire code base becomes difficult as updating takes longer than the time between two subsequent commits to the repository.

While other code search systems focus on text-based search or use test-cases to find suitable code for reuse, we build on the approach of using a method's context to find useful methods for reuse. We complement this approach with signature matching and techniques known from software engineering: architecture analysis and code review states of code. These are used as metrics to rate a method's suitability in a given context.

This thesis contributes a novel code search system designed for use with companies' internal code bases. The proposed system uses an incremental approach to update its index, updating only parts of data affected by the changes in each commit. This keeps the analysis time low and allows the system to work with updated data in the matter of seconds.

To evaluate the proposed code search system a novel evaluation method is introduced that is built on the concept of incremental analysis. It enables the usage of a system's entire development history, accurately replaying the actual development process. The system's state and the changes made at any given commit can be used to accurately evaluate the code search system using the data that would have been available at that time.

The evaluation shows that our code search system consistently delivers results containing the developers choice on two different systems in more than 50% of the cases when retrieving 5 results. We evaluate all combinations of our proposed metrics and show which lead to the best results.

Acknowledgements

First I want to thank Prof. Dr. Dr. h.c. Manfred Broy for giving me the opportunity to write my thesis at his chair and for supervising my work.

I am grateful to Veronika Bauer, Dr. Lars Heinemann and Dr. Elmar Jürgens for supervising my thesis and providing invaluable support and guidance for my work and research. Without their feedback, support and ideas this thesis would not have been possible.

For giving me the opportunity to work with them as part of my thesis, giving me technical support whenever necessary and providing me with a great place to work on my thesis I am thankful to *CQSE GmbH* and all its employees.

My gratitude goes to my friends, especially Anna Borgmann, Moritz Beller and Fabian Streitel who have been incredibly supportive during the entire time. Without their ideas, help and cheering up the writing time would not have been as joyful.

I am thankful to Dr. Lars Heinemann, Veronika Bauer, Fabian Streitel, Moritz Beller, Anna Borgmann, Lukas Küpper, my parents Andreas and Gabi and my Sister Johanna Kinnen for proofreading my thesis and significantly improving its quality through their hard work.

Thank you to Moritz Beller and Fabian Streitel for your detailed corrections in my Bachelor's Thesis and pointing out its most repetitive mistakes. It's through your hard work that I learned to avoid them in my Master's Thesis. Therefore, I am ever grateful to both of you.

Last but not least I want to thank my family for supporting me on the way through my studies to this thesis and during its writing. Without you it would not have been possible to complete my studies and start into the working life equipped with the necessary technical and social skills.

Table of Contents

1 Introduction

1.1 Motivation

A developer working on software frequently faces the question: "Has someone written the code that I am about to program?". While this question can often be answered fairly quickly on small projects, it is difficult to answer on industrial scale projects with hundreds of thousands of lines of code. The programmer can no longer read all the code to find possible sources to reuse but has to resort to using his experience or tools to search the code base.

As software development is a collaborative effort on bigger projects, it becomes increasingly difficult for a developer to track all the changes made by his colleagues. Google for example has more than 20 changes per minute[1]. Keeping track of all new code would be desirable in order not to miss future reuse opportunities but becomes impossible on projects with many contributers as the amount of produced code cannot be memorized by one single individual.

In software engineering reuse of existing software has been associated with many benefits: It shortens development cycles, improves code quality and reduces the amount of code to be maintained [20, 26].

Many forms of reuse have emerged since it was first envisioned in 1968 by McIlroy [28]. The simplest form of reuse is copying and pasting (cloning) code snippets from other files or programs. This is generally viewed as bad practice and is known to cause bugs [22].

Reusing existing methods or entire classes as a whole is an established standard in most software projects today [17]. The reused methods and classes can be part of the project's own code or contributed by external libraries. Using external libraries has the benefit of being provided with generally well tested and commonly used code. Thereby automatically improving the code's quality. The reoccurrence of errors that have already been found and corrected by the library's authors can thus be prevented. Consequently, the programmer reusing the code saves time and money.

With the rise of component-based and service-oriented development paradigms it has become popular to reuse entire components instead of only smaller parts like methods and classes. The reuse of larger components requires a deep understanding of their behavior in order to correctly evaluate if they solve the problem at hand and its particular conditions. Many code search engines provide service or component search (e.g. *Merobase*[2], *Service Repository*[3]).

[1] Google Engineering Blog: `http://google-engtools.blogspot.de/2011/05/welcome-to-google-engineering-tools.html` (last accessed October 14, 2013)
[2] Merobase Component Finder: `http://www.merobase.com` (last accessed October 14, 2013)
[3] Service Repository: `http://www.service-repository.com` (last accessed October 14, 2013)

In recent years the open source community has seen a huge rise in the number of projects [11], making it a valuable source of reusable software. Open source libraries like the *Apache Commons*[4] have become popular even among industry projects, as they often provide sources of well tested code suitable for use in production quality software. To aid the reuse of open source software, multiple source code search engines have been introduced, e.g. *Krugle*[5] and *Ohloh Code Search*[6]. The latter indexes over 20 billion (October 2013) lines of code (LoC) that can freely be searched for any text. Nevertheless, research indicates that developers often do not find what they are searching for using services like *Ohloh Code Search* [3].

1.2 Problem Statement

As code bases grow, it is impossible for a single developer to know all the existing functionality that is implemented within an application or library. This leads developers to implement functionality that is available as part of the code base and that could be reused. Every time a reuse opportunity is missed, an increased maintenance burden and inconsistencies in the code base are likely to follow [23, 31, 22].

To aid developers with this problem, a system is needed that can assist them in finding reusable code snippets in the entire code base. As companies tend to have an ever growing collection of projects and code, it can easily happen that developers does not have the company's entire code base on his local machine, rendering local text search methods like *grep*[7] or the *Windows search*[8] useless for finding reusable code.

Using Internet scale code search engines for internal code search does not seem to solve the problem because the presented results appear to be inaccurate in many cases [3]. Other approaches are needed to sort the presented data, as not all code is suitable for reuse and thus of limited interest to the developer.

Regular code search methods ignore additional information, such as the software's architecture or the code's review status. Using a method from a package whose usage is forbidden by the architecture leads to inconsistencies between the code and the architecture. This possibly makes a reuse candidate a worse choice for than one that is allowed. Reusing code that has not yet been reviewed comes with the risk of using code that is still under development and possibly contains more bugs. Instrumenting the review ratings to ignore code that has not yet been reviewed in the search could save the developer from these risks.

[4] Apache Commons: `http://commons.apache.org` (last accessed October 14, 2013)
[5] Krugle: `http://krugle.org` (last accessed October 14, 2013)
[6] Ohloh Code Search: `http://code.ohloh.net` (last accessed October 14, 2013)
[7] grep website: `http://www.gnu.org/software/grep` (last accessed October 14, 2013)
[8] Windows search website: `http://windows.microsoft.com/de-de/windows7/products/features/windows-search` (last accessed October 14, 2013)

1.3 Contribution

This thesis contributes a new approach to search searching for reusable methods by using architectural information (section 2.9) and review state of code (section 2.3). These are used as additional metrics to context matching (section 2.8) and signature matching (section 2.4) for the retrieval of reusable methods.

We present a working prototype implemented as part of *Teamscale* (see section 2.11). It is built on the usage of a text indexing framework as search back end for data retrieved. Our implementation provides a service accessible to clients like our proposed IDE auto-completion system, which formulates queries based on the programmer's current context and makes the results available to the developer without any manual interaction. Due to our system's incremental approach we provide up to date information in a matter of seconds after each commit.

Additionally, we propose an incremental approach that evaluates our system using a study object on a per-commit basis, therefore closely following the actual development process. We measure the effectiveness of our approach on different study objects (see chapter 5). Our evaluation approach is useful for all recommendation systems applied in an auto-completion scenario where the code's history is available.

1.4 Research Questions

We propose a set of research questions.

RQ 1: What are useful metrics and techniques for a code search system?

We want to identify a set of metrics that are useful in our scenario of a closed, company internal software project to determine reusable methods, so that helpful results can be presented to developers by a code search system.

RQ 2: Which combination of metrics provides the best results?

The proposed metrics should be evaluated and tested in multiple combinations on real systems to find out which combination provides developers with the best set of reusable methods for their query to the system.

RQ 3: Can a shallow parser replace a full parser in code search engines?

In order to generate data for our code search system we have to inspect the source code of the underlying project. Our aim is to provide a maximum of indexing performance, so that the latest data can be made available to the back end as quickly as possible. A shallow parser can read great amounts of code quickly and provides a level of programming language independence through abstraction. As this comes at the cost of limited functionality, we want to evaluate whether using a shallow parser could be a valid option compared to a full parser.

2 Preliminaries

This chapter briefly introduces techniques used in this thesis and explains basic concepts needed to understand it.

2.1 Auto-Completion in the context of an IDE

Most modern IDEs (integrated development environment), like *Eclipse*[1], *Netbeans*[2] or *IntelliJ IDEA*[3], provide developers with a way of automatically completing the word they are currently writing. This can mean completing the name of a method, variable, a class name, language keywords or more advanced use cases like automatically inserting a while loop with all necessary syntax. Generally, the results are sorted by expected return type and then lexicographically taking the already written characters into account. Figure 2.1 shows an example of the *Netbeans* auto-completion.

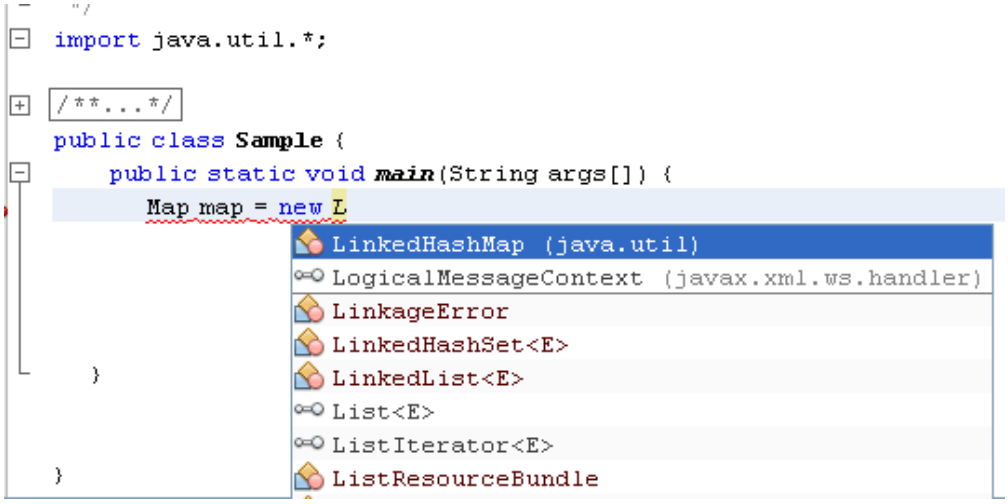

Figure 2.1: A sample of a *Netbeans* auto-completion session [4]

[1]Eclipse website: `http://www.eclipse.org` (last accessed October 14, 2013)
[2]Netbeans website: `http://www.netbeans.org` (last accessed October 14, 2013)
[3]IntelliJ IDEA website: `http://www.jetbrains.com/idea` (last accessed October 14, 2013)
[4]Taken from `https://netbeans.org/kb/docs/java/editor-codereference.html` (last accessed October 14, 2013)

2.2 Internet Scale Code Search

The open-source movement has made large software projects freely available on the Internet. Similar to searching the Internet for websites with Google or other search engines, researchers and companies [2, 18, 19] found great interest in building search engines used to search for source code on the Internet.

Internet scale code search engines usually index up to billions of lines of code (e.g. *Ohloh Code Search*[5] - $\geq$ 20 billion lines of code) and make them available through a search interface. Many code search engines like *Ohloh Code Search* work just like normal text retrieval engines and provide full text search. Others use more abstract data, e.g. the *Merobase Component Finder*[6], which allows the user to specify tests in order to find entire matching components.

2.3 Code Reviews

Code review is a process in which a developer systematically examines source code to identify problems. Code reviews were introduced in 1976 by Michael Fagan under the name of formal inspections [12]. The inspections include the review of design, tests and documentation. Code reviews can have multiple purposes, such as identifying actual bugs, finding code with an inconsistent style or locating maintainability problems.

2.3.1 LEvD Code Review Rating Process

The *CQSE GmbH* employs the *LEvD* review process [10] to review code written for *Teamscale*. It is a light weight review process tightly integrated with a bugtracker. Every file can be in one of three basic states: RED, YELLOW and GREEN.

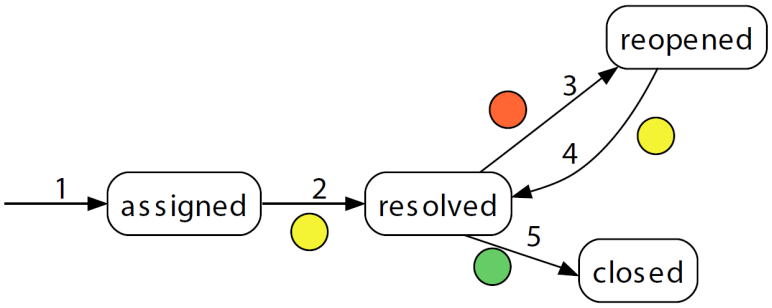

Figure 2.2: The LEvD review process.

Figure 2.2 shows an activity diagram of the process. The process starts with a ticket (bug report, feature request on the bugtracker) *assigned* to a developer and separate QA contact, who is in charge of the review. The developer then works on his changes

[5]Ohloh Code Search: `http://code.ohloh.net` (last accessed October 14, 2013)
[6]The Merobase Component Finder: `http://www.merobase.com/` (last accessed October 14, 2013)

and before committing the code back, marks the code YELLOW and sets the ticket to *resolved*. The QA contact then reviews the code. He marks all files that do not need further improvement GREEN. Files that need improvements are marked as red and review comments are added using *TODO* statements in the code.

In case all files under review are marked GREEN the ticket can be *closed*. If any file is left RED, the ticket is set to *reopened* and the developer has to revisit any changes requested by the reviewer. After he is done performing the changes and set all files back to YELLOW, he sets the ticket back to *resolved*. The cycle can then be repeated until the reviewer does not find any more necessary changes and marks the code as GREEN and sets the ticket to *closed*.

The *LEvD* review process is used in *ConQAT*, on of our study objects, and thus has to be supported by our code search system in order to use the review states.

2.4 Signature Matching

Signature matching is a technique introduced and defined by *Zaremski* and *Wing* [35]. It determines whether two methods are syntactically interchangeable.

The signature for every method is defined by its return value and argument types. Matching signatures indicate that methods are syntactically exchangeable, but they do not guarantee that they have the same semantic functionality.

For example a simple Java method signature (listing 2.1) consists of a return type (here: *boolean*) and argument types (*String, int*). The argument names (*bookName, Year*) and method name (*existsBook*) are not taken into account for the matching process, as they can easily be renamed.

Listing 2.1: A basic Java method signature

```
1 private boolean existsBook(String bookName, int year);
```

A more abstract notation of the *existsBook* method's signature, as used by *Zaremski* and *Wing*, is given in listing 2.2. The left part contains the argument types, while the right-hand side of the arrow indicates the return value. The method has been abstracted to contain only the information used for signature matching.

Listing 2.2: Abstract ML style signature description

```
1 (String, int) → boolean
```

When searching for methods in source code, one can use all or single parts of the signature to find matches.

Zaremski and *Wing* defined a number of strict and relaxed matching rules:

1. *Exact Match*: Matches only if return type and argument types strictly match.
2. *Generalized Match*: Matches if types in the signature database are more general than the ones given in the query.
3. *Specialized Match*: Opposite of generalized match. Matches for specialized types.
4. *Generalized and Specialized Match*: A combination of the two allows more general as well as more specialized types to match.
5. *Unify Match*: Matches if types in the query can be unified to match a signature in the index.
6. *Reorder Match*: Allows argument types to be swapped in order to match the query.

For exact definitions and a more detailed explanation refer to [35].

In theory, it is possible to use signature matching on its own to search for methods. One could, for example, try to find matching signatures for $(String,\ double) \rightarrow boolean$. Doing so requires the developer to specifically know what he needs. Additionally, it ignores all information that is encoded in the method name and argument names.

We use signature matching in our code search engine to narrow down result lists and to provide functionality that can be useful in an auto-completion scenario.

2.5 Reusability Metrics

A number of reusability metrics for methods, classes or entire components have been proposed [30, 33]:

- McCabe Cyclomatic Complexity [27]
- Halstead's Software Science Metrics [14]
- Barnard's reusability metric for object-oriented software [4]

A reusability metric is meant to allow an objective answer to the question: "*How reusable is software X?*". As discussed in [30], the answer to this question is dependent on the definition of reusability by the individual researcher or developer and is thus difficult to answer universally.

2.5.1 Number of Method Invocations and other Code-Completion Metrics

The *Eclipse Recommenders Project*[7] started with the approach of counting the number of times a method is invoked from a given code base. The idea being that methods that

[7] Eclipse Recommenders Project website: `http://www.eclipse.org/recommenders/` (last accessed October 14, 2013)

have been called many times should be more reusable than those rarely called because they have been reused multiple times in other situations.

Bruch et al. [7] found this approach to be quite limited for the prediction of which methods of a given class the developer was looking for. However it can be used as a good baseline to evaluate other approaches: Any newly introduced approach should be better than the "most used methods" approach.

Bruch et al. proposed two other metrics: *association rule based code completion* and *best matching neighbors code completion*. They found the *best matching neighbors* algorithm to work best in their test scenario and implemented it as part of the *Eclipse Recommenders Project*.

2.6 Shallow Parsing

Shallow parsing is a technique to parse source code on an abstract level. A shallow parser only provides information about imports, classes and methods, but not more details like type hierarchies and types for variables or method arguments.

This allows the program to understand the basic structure of a source file, without having to use a full parser/compiler. If the programmer needs to retrieve additional information about the source code, which is not provided by the shallow parser, he can fall back to using the underlying token stream provided by the lexer. Often times this is powerful and simple enough to opt for not using a full parser.

Because of their relative simplicity, when compared to full parsers, new shallow parsers can be constructed fairly quickly and can parse large amounts of source code in a short time.

The *ConQAT*[8] toolkit provides shallow parsers for many languages including *Java, C#* and *C++*. The different shallow parsers are wrapped behind an abstract interface, allowing the same program to work on different programming languages without any change, as long as one does not work with the token stream.

We use *ConQAT*'s shallow parser framework with the goal of creating a code search system that is as language agnostic as possible, in order to make the system available for many languages.

2.7 Latent Semantic Analysis (LSA)

LSA [8] is a technique used to analyze relationships between a set of documents, like a set of documentation texts, by automatically creating a set of concepts related to the documents and the terms they contain. The assumption is that similar texts will contain words from similar concepts. It is therefore useful to find similar text pieces, given a set of words or an entire text. This concept is used by *Context-Dependent Method Recommendation* (section 2.8) to find similar contexts.

[8] *ConQAT* website: `http://www.conqat.org` (last accessed October 14, 2013)

2.8 Context-Dependent Method Recommendation

Context-Dependent Method Recommendation [16] is a technique that uses the text around method calls (context) to find reusable methods by matching their contexts to new contexts by defining a similarity between them. The context of a given method call in the source code can be the preceding N identifiers, or the surrounding lines. Other options like using the method call's position in the document, e.g. is the item in a class-, method- or if-statement, etc. are possible as well. For this thesis, we will be be referring to the *context* as the last N identifiers before an item.

We define the *lookback* as the number of identifiers we collect preceding the item.

Which identifiers are collected depends on the specific implementation. It is common to leave out basic operators or even keywords, as one could argue that they do not add value to the context. This is similar to the concept of stop words used in many text search engines. A stop word is a word that appears so often, that it does not provide any unique information. Typical examples in the English language are: "or", "is" or "and".

Listing 2.3: Java context example

```
1 private double calculate(double lower, double ceiling) {
2   double average = (lower + ceiling) / 2;
3   return min = Math.min(lower, average);
4 }
```

For example, if we collect the context for $Math.min(lower, average)$ given in listing 2.3 and use a *lookback* of 4, we get: $[lower,\ ceiling,\ 2,\ min]$.

This context extraction can be done for all method calls in the entire code base, generating a many-to-one mapping of context sets to method calls. Given a new context C, one can compare it to the already collected contexts using similarity metrics like *LSA* (see section 2.7). A method call with contexts similar to the new context C may indicate that the called method could be useful at the new contexts position as well.

Research performed by *Heinemann et al.* [16] shows that a *lookback* of 4 leads to the best results on average. Therefore we will be using this value for our work.

2.9 Architecture Analysis

Most bigger software systems have an architecture description that explains which parts of the software are allowed to interact with each other and which are not. It also describes the intended interactions between the different components.

To avoid possible decay through an architecture's life-time, architecture conformance assessment has been proposed [29]. An architecture analysis calculates the real architecture of a software system by collecting dependencies between different parts and compares them to a specification. It can then detect illegal accesses that are found in the code (or other subsystems like the database) and bring them to the developer's attention.

A flexible framework for architecture assessment has been proposed [9] and implemented in *ConQAT*.

"ConQAT's architecture model comprises components and policies. Policies model allowed and denied accesses between components. Each component must have either a subcomponent or a codemapping. A codemapping is a regular expression that defines which types from the source code belong to the component the codemapping is associated with." [6]

Figure 2.3 shows a basic architecture for the Java project JUnit[9]. The image illustrates how components like *Core*, *Lib*, etc. are connected by policies indicating access rights. For example code from the *Extensions* component is allowed to access code in the *Framework* component, but not the other way around. This information is stored in the policies connecting components, indicated here by the green ('*ALLOW*') and red ('*DENY*') arrows between different components.

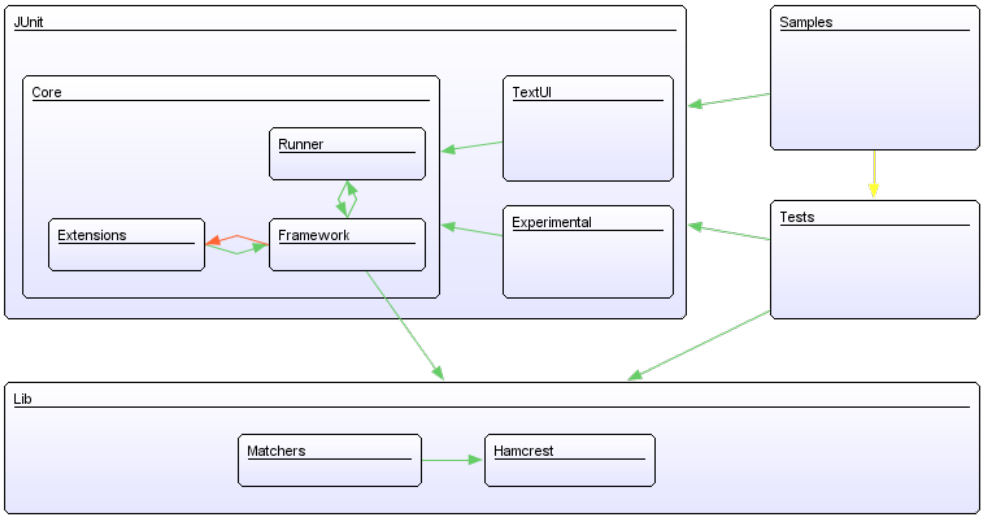

Figure 2.3: A basic JUnit architecture in *ConQAT*[10]

A yellow arrow indicates a '*TOLERATE*' policy, which contains a number of architecture violations which are accepted during the analysis. This is useful to ignore older violations until they can be addressed, while still making sure that no new violations are added during development.

To perform an architecture conformance analysis the actual architecture is extracted from the source code or other artifacts, by creating a dependency graph. For source code it can be created by looking at e.g. imports or method calls. Other artifacts, like databases, need special definitions for access to other modules, so that they can be recorded.

The dependency graph is then mapped onto the reference architecture by using the software reflexion model technique [29], validating all policies. The result is a list of legal and illegal accesses, showing the project's conformance to the reference architecture.

[9] JUnit Website: `http://www.junit.org` (last accessed October 14, 2013)
[10] JUnit architecture from: `https://www.conqat.org/demos/tutorial_architecture/junit/` (last accessed October 14, 2013)

2.10 Lucene Search Engine Framework

Lucene[11] is a Java framework originally created for building text search engines. It provides powerful query parsing and customization capabilities.

In *Lucene* the basic searchable unit is a *Document*. This can be a text document, a website or anything else to which textual information could be attached (e.g. methods in our case). A document is made up of multiple fields which can hold data in different formats (text, unique identifiers, numbers, etc).

There are three basic types of fields: *StringField*, *TextField* and *IntField*. *StringField*s hold identifiers that will not be split into multiple terms, whereas *TextField*s contain text that is split into terms and analyzed. This allows substring queries when using an analyzer which supports this. An *IntField* contains a number, allowing range queries.

Each field has an assigned analyzer which processes the field's content into multiple terms and is able to do post processing on the them (e.g. lowercase everything). A *term* is the smallest *Lucene* unit, which often contains exactly one word.

We use the following analyzers[12]:

- KeywordAnalyzer: Stores the content as-is, useful for keywords
- WhiteSpaceAnalyzer: Splits the given input string by whitespaces
- SimpleAnalyzer: Splits the input at non letter characters and lowercases everything
- StandardAnalyzer: Normalizes tokens, splits words based on a grammar, removes stopwords and lowercases everything

Lucene stores a term vector (called *TermFreqVector* in *Lucene*) to allow fast lookup of documents containing specific terms. It includes a mapping of terms to documents and their occurrence frequency.

A query parser is provided by the framework. It can parse complex queries to the search engine and allows combining queries with AND/OR and assignment of different weights to separate parts oft the query. The analyzers are used to process the query for each different field, each returning a list of used terms. From this list the term frequency vector is calculated, which makes the efficient lookup of search results possible.

We use *Lucene* as our search back-end framework for our code search engine (see section 4.3.2).

Example

We want to index the set of supermarket products shown in table 2.1. For each product we have a unique ID, a name and a description.

[11] Lucene website: `http://lucene.apache.org` (last accessed October 14, 2013)

[12] Lucene Analyzers Doc.: `http://lucene.apache.org/core/4_3_1/analyzers-common` (last accessed October 14, 2013)

Id	Name	Description
ID12345	"Chocolate Cake"	"Chocolate cake that tastes fluffy and jummy."
ID56789	"Cookies Deluxe"	"Cookies made with chocolate and milk."
ID98765	"Coke"	"Sparkling drink with caffeine."

Table 2.1: Example supermarket products that are to be indexed in *Lucene*.

We have to model the document to reflect the data and select appropriate analyzers to work with it. The document will have three fields: *ID*, *Name* and *Description*. Each field needs different analyzers to work as intended.

As we only want to find products by ID if the entire ID matches, we choose the *KeywordAnalyzer* for the *ID* field, the type of this field is a StringField, as no splitting and deep analysis is done. It will index the entire ID as one term. If the product name consists of multiple words, we want to be able to find it using only a subset of these. We therefore choose the *WhiteSpaceAnalyzer* for the *Name* field. The *Name* field is set to be of type *TextField* because the contained text should be analyzed and indexed in a way that allows us to find substrings. The *Description* is set to use the standard analyzer, as we want the capabilities of full-text search for this field. A summary graphic of our document model is given in fig. 2.4.

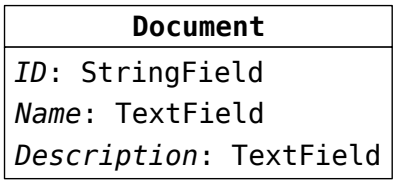

Figure 2.4: Our example document structure

Each product we want to enter into the search engine is added by creating a document for it and filling all fields with data. Next the *Lucene* analyzers do their job and extract the terms to make the product findable.

After the index is filled with documents it can be queried for matching results using the *Lucene* query language[13]. If for example we want to search for any cake in the index, we could use the query: "*Name:'Cake' OR Description:'cake'*". This would run the strings in the query through the respective analyzers for each field and try to match them with the content in the index. In our case it would check if the word 'cake' was to be found either in the name or description field and would return "Chocolate cake". Searching for "*Description:'chocolate'* would return both the "Chocolate Cake" and the "Deluxe Cookies", but not "Coke".

[13]Lucene Query Language Overview: `http://lucene.apache.org/core/4_4_0/queryparser/org/apache/lucene/queryparser/classic/package-summary.html#Overview` (last accessed October 14, 2013)

2.11 Teamscale

Teamscale[14] is a code quality monitoring system developed by *CQSE GmbH*[15]. It is based on the open-source quality analysis toolkit *ConQAT* which provides means for running quality analyses and creates dashboards presenting the collected results.

Teamscale reads data from a revision control system and performs quality analyses on the code after each commit. All analyses are implemented using an incremental approach which only updates the database with the changes from every new commit, without reanalyzing the entire repository. *Teamscale* allows access to the collected data through a wide range of web services that can be used by different clients, e.g. the *Teamscale* web UI or IDEs like *Eclipse* or *Visual Studio*.

Teamscale's architecture (fig. 2.5) [5] is explained in the following paragraphs.

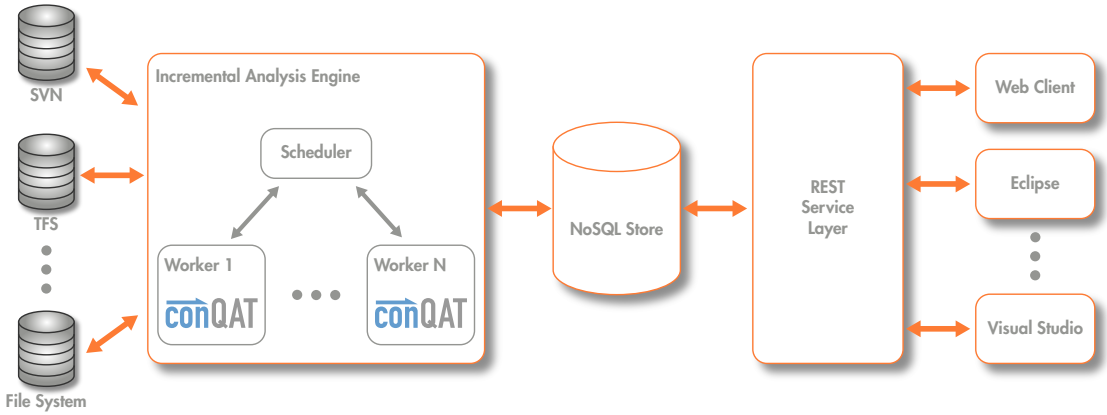

Figure 2.5: Teamscale architecture

Storage System To store processed data in *Teamscale*, a key/value storage is used (*NoSQL Store* in fig. 2.5). It allows efficient queries using single keys, key ranges and key prefixes. The storage layer can utilize a variety of NoSQL databases as its back end.

The key/value storage has a simple interface to manage data using byte arrays as keys and values. The interface is depicted in fig. 2.6. Three basic operations are supported, *put* for adding data to the store, *get* for data retrieval and *remove* for data deletion. Because using byte arrays is a very low level of access, *Teamscale* provides multiple wrappers (e.g. to use Strings instead of bare byte arrays as keys and values) around the basic interface.

Trigger An analysis run in *Teamscale* is defined by a set of *triggers*. Each trigger describes one step of the analysis and can be run periodically or only when the required input data changes.

[14]Teamscale website: `http://www.teamscale.com` (last accessed October 14, 2013)
[15]CQSE GmbH website: `http://www.cqse.eu` (last accessed October 14, 2013)

```
IStore
byte[] get(byte[] key)
void put(byte[] key, byte[] value)
void remove(byte[] key)
```

Figure 2.6: The basic interface used by *Teamscale* stores.

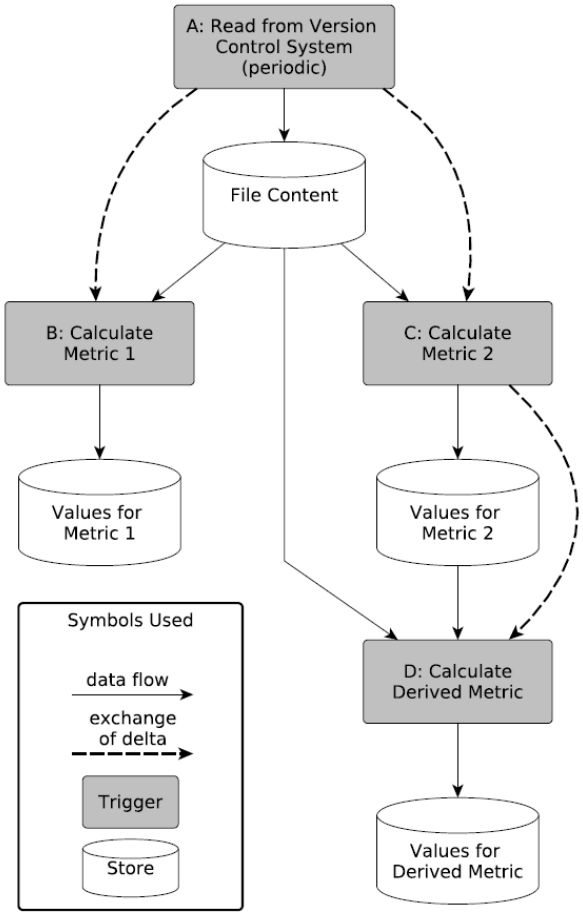

Figure 2.7: An example *Teamscale* configuration

Job Scheduling A job is defined as an analysis that is to be run on a set of files. It has two input parameters: a set of modified keys that should be analyzed and the trigger that is to be run. The job writes its output to the storage system and the changes made are recorded. This allows running other jobs based on the changed data of a previous job. A basic example of reading data from a revision control system and running a set of triggers to calculate metrics is shown in fig. 2.7.

In step (A) data is read form the revision control system. The read data is stored into

the *File Content* storage and the changed keys (also called delta) are passed to the metric calculations in *B* and *C*. These can access the files' content by using the *File Content* storage. Next the calculated metrics are saved into a storage specific to each metric. The values calculated in *C* are then used to calculate a derived metric in *D*, which uses values from *C* and files from *A*.

Historization *Teamscale* only stores new values for changed files into the stores. Values for unchanged files are stored for the old version. This allows the retrieval of the entire set of values for every given point in time, which is useful for calculating trends and metrics that make use of history information.

2.11.1 Metric Types

There are two types of metrics: *local* and *global*. *Teamscale*'s architecture allows the calculation of both metric types.

Local Metrics Metrics that can be calculated using only a single file are called *local*. Given a set of keys from the storage system that should be checked (e.g. list of files changed in a commit), the metric can be calculated without changing values for files outside the given key set. Typical examples for this type of metric are lines of code or method length.

Global Metrics Metrics whose calculation possibly affects values of unchanged files are called *global*.

One example for this type of metric is the clone coverage. A clone is a piece of source code that appears at multiple positions in the same or multiple files [23]. The clone coverage is the ratio of lines covered by at least one clone [21]. As a clone can appear in multiple files, analyzing a modified file possibly finds clones with other files, thereby changing their respective clone coverage.

3 State of the Art

In the following chapter, we will give a short overview of the current state of the art in code search engines and tools.

Code search systems aim at helping the developer find code that he can reuse for a problem he is currently facing. If the system can efficiently help developers find what they are looking for, then there is a potential of saving time and money through reuse of existing code.

All systems that are used to search through code bases to find pieces of source code are called code search systems. In their most basic form these can be general purpose text search tools like the Unix utilities *grep*, *find* or the *Windows Explorer* search functionality, even if not specifically designed for the special purpose of searching through code. They allow the user to specify keywords that should be matched in the searched files, highlighting any matches for the user.

As developers are facing ever growing code bases with millions of lines of code and the amount of publicly available code on the Internet keeps growing, researchers and companies alike are looking for more efficient ways of searching through code. Recent systems are usually more specialized tools, e.g. an Internet search website dedicated to searching code from open-source repositories (see section 3.1) or services providing special APIs which can be queried using complex query languages or even test cases. This chapter will give a detailed overview of code search systems available today.

3.1 Internet Scale Code Search Engines

In this section, we introduce the important available Internet scale code search engines available today.

Internet scale code search engines index code found on the Internet and contain millions of lines of code. They work like normal text based website search engines (e.g. `http://www.google.com`) with a search field for full text search. The user can specify a number of keywords to retrieve pieces of source code in which these keywords are found. Many engines provide special keywords to search specifically for method names or class names, allowing the developer to be more precise in his query.

3.1.1 Ohloh Code Search

Ohloh Code Search[1] is the biggest freely accessible code search engine. With over 20 billion lines indexed it provides (October 2013) the largest index of open-source source-code

[1] Ohloh Code Search website: `http://code.ohloh.net` (last accessed October 14, 2013)

available on the Internet. On October 25th 2012 Koders[2] was merged into Ohloh's code search engine.

It allows full-text search and provides advanced search functionality for method/-class/structure definitions using special keywords like $cdef :< searchterms >$ for class names. *Ohloh Code Search* does not provide advanced matching functionalities like signature matching, and cannot make use of project specific information like architecture information to enhance its results.

Ohloh Code Search is powered by the commercial code search engine *Code Sight* developed by *Black Duck Software*[3].

3.1.2 Krugle

Krugle's OpenSearch[4] currently indexes around 350 million lines of code, most of which is written in Java. It allows full-text search of the indexed source code, as well as some advanced searches for method definitions, method calls and class definitions.

It is very similar to *Ohloh Code Search*, but smaller in size. Like *Ohloh Code Search* it does not provide any advanced matching functionalities and does not have the ability to consider project specific information like the architecture or code review ratings.

3.1.3 Sourcerer

Sourcerer[5] [2] is a code search engine specifically for *Java*, that was developed to be a back-end for external clients by the *University of California, Irvine*. It provides an in-depth model of the imported projects and stores the relations between source files and entire projects and their dependencies. Additionally, it comes with a signature matching service and index, as well as a web-service allowing access to the source code through an internal file repository.

The *Sourcerer* project consists of multiple components. The lowest layer component is the core infrastructure layer, which is used to crawl the Internet for source code, process and index it. The downloaded source code is stored in a second component, the repository. The repository was made available to the research community, but is currently unavailable due to the repository service being off-line (October 2013). Using the information stored in the repository, the core component extracts structural information from the source code and stores it into a third component: *Sourcerer DB*. This relational database can be used as the base for clients and is made available for read-only access to the researching community.

The first client to be developed on the base infrastructure offered by the *Sourcerer* project is a code search service. The code search service provides a query interface that

[2] Website formerly: `http://www.koders.com` (last accessed October 14, 2013)
[3] Code Sight website: `http://www.blackducksoftware.com/products/code-sight` (last accessed October 14, 2013)
[4] Krugle OpenSearch: `http://opensearch.krugle.org` (last accessed October 14, 2013)
[5] Sourcerer website: `http://sourcerer.ics.uci.edu/index.html` (last accessed October 14, 2013)

can be used with *Lucene* queries, allowing complex queries on the entire database. A web interface is provided for easy use. The test-driven code search tool *CodeGenie*[25] was built on top of the code search services.

The *Sourcerer* project provides a complex infrastructure indexing open-source code on the Internet and providing access to it via web services. As *Sourcerer* is a large scale search infrastructure, it does not provide means for the application of project specific information like architecture information and review results. Because *Sourcerer* scans many sources for changes, not all changes submitted to these sources will be available in a matter of minutes, which might possibly result in out-of-date search results.

3.2 Component Search Engines

Component search engines work on a more abstract level than code search engines. They allow the user to search for specific interfaces or functionality by other means than just full text search. For example they can offer test based search or use some form of formal query language enabling the user to search for very specific functionality.

3.2.1 CompRE

CompRE [1] is based on an ontology based model called SCRO (Source Code Representation Ontology). It captures object oriented dependencies between artifacts like inheritance, method overriding, method overloading and method signature information[6].

To save manual annotations and semantical information, the SCRO model is enhanced to contain domain specific information like a description or information about the input parameters. This extended SCRO model is called COMPonent REpresentation ontology (COMPRE).

The ontology model is built automatically by analyzing the source code, but the domain specific knowledge that is annotated needs to be entered manually into the knowledge base.

To demonstrate the capabilities of the model an Eclipse plug-in called *CompRE* was implemented. It allows the creation of complex queries for three scenarios:

- Type or signature based queries
- Metadata keyword queries
- Pure semantic-based queries
- A combination of the three

Data is stored using the *Lucene* framework (See section 2.10). All queries in *CompRE* have to be provided using the *Lucene* query language. *CompRE* does not provide any automated recommendations based on the current context. Thus, every developer has to

[6]SCRO website: `http://www.cs.uwm.edu/~alnusair/ontologies/scro.html` (last accessed October 14, 2013)

learn how to use the query language to use *CompRE*. Additionally, the index has to be manually annotated and cannot be generated fully automatically, making it a slow and labor intense process.

3.2.2 Merobase & CodeConjurer

Merobase[7] has all features of a typical source code search engine, but also includes the ability to search for component interfaces, web-services or even compiled source code found on the Internet. Components can be found by using specialized commands for the full text search or by using test-driven search.

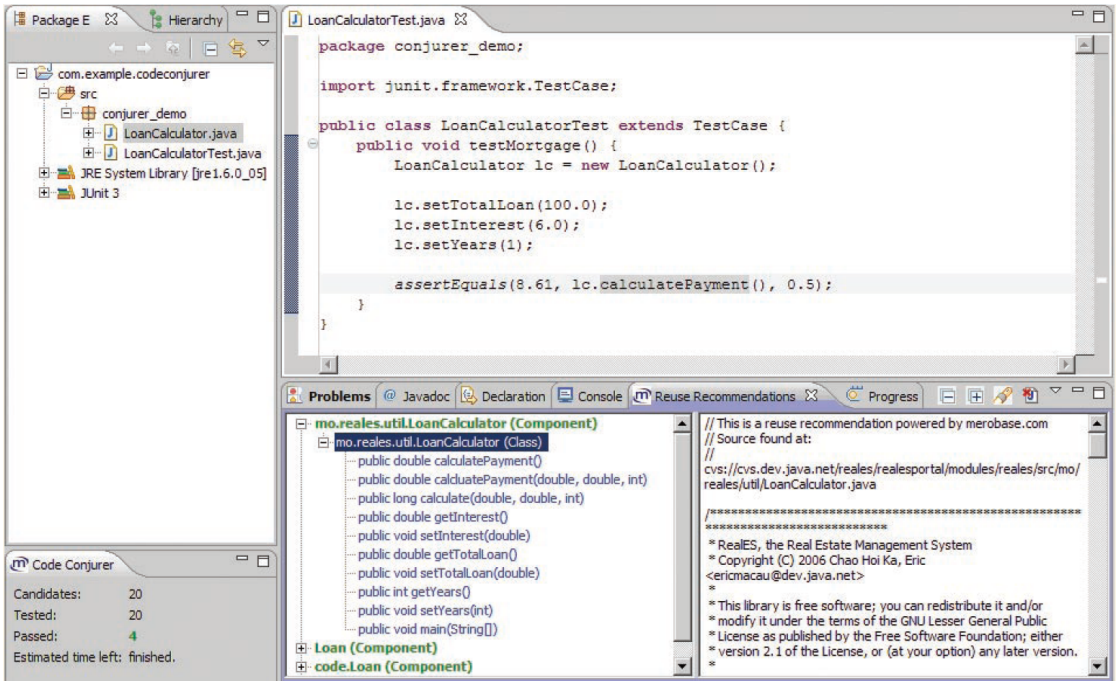

Figure 3.1: The *CodeConjurer* main view, demonstrating a test case and the corresponding results. Taken from [19]

.

CodeConjurer [19] is an Eclipse plug-in that uses *Merobase* as a back-end and allows the developer to formulate test cases, which he wants the retrieved components to pass. By specifying test cases (see fig. 3.1 for an example), the developer can not only make sure the interface matches his needs, but can also specify the behavior of the component in question. To find components with the correct behavior *CodeConjurer* tries to compile the components and places them into the specified test case. Next, the test cases are run. The components that fulfill the most test cases rank highest.

[7] Merobase website: `http://www.merobase.com` (last accessed October 14, 2013)

3.3 Code Recommenders

A code recommender is a tool that provides code reuse proposals based on the current source code position, without the user's direct help. It autonomously collects the information needed to query the used index and constructs a query to find useful results for the developer. The results are displayed to the user without further interaction required.

3.3.1 CodeBroker

CodeBroker [34] aims to find reusable code when the programmer programs a new method. It analyzes the method's documentation string and method signature. It is implemented as an Emacs[8] plug-in.

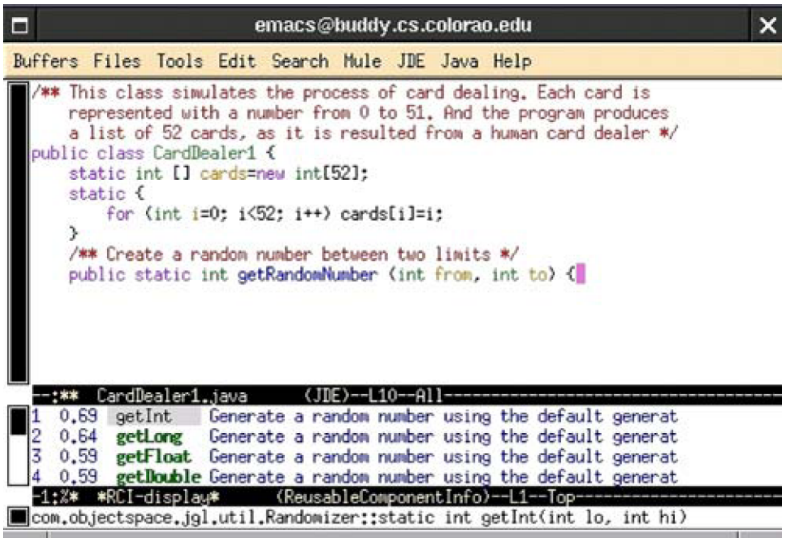

Figure 3.2: CodeBroker's main view. Below the editing area, recommended source code samples are displayed. In this case the developer is trying to implement a method that returns a random number in a given range.

In the example given in fig. 3.2 the developer's goal is to implement a method that creates a random number in a given range. *CodeBroker* now analyzes the documentation text and compares it to the index of method documentation texts in the index, using *Latent Semantic Analysis* (section 2.7).

CodeBroker employs this idea to find methods with a related documentation text, given the newly written documentation provided by the programmer. In case the programmer also provides a method signature, it is matched against all methods in the database using method signature matching as introduced in section 2.4. The results from LSA and signature matching are combined and ranked in order to provide the best possible result to the developer.

[8]Emacs website: `http://www.gnu.org/software/emacs/` (last accessed October 14, 2013)

Finally, the results are then displayed in the lower part of the screen, making them available to the developer. By running the queries automatically, *CodeBroker* ensures that the developer is not distracted from his work and is presented with results when he needs them.

3.3.2 Eclipse Code Recommenders

Eclipse Code Recommenders is an Eclipse[9] plug-in which changes how the Eclipse auto-completion sorts its results. Instead of sorting them lexicographically they are sorted by their frequency of use in the current context. To achieve this, the project analyzed a number of open-source projects' source code and extracted a model containing records of how often every method is used. The idea is that methods that have been used many times are both reusable and useful to the programmer, as proven by their frequent use.

Especially in the Eclipse APIs, many classes contain over 100 methods a few of which are actually employed most of the time. These are displayed first when triggering the auto-completion in the expectation that the programmer will need them (See fig. 3.3).

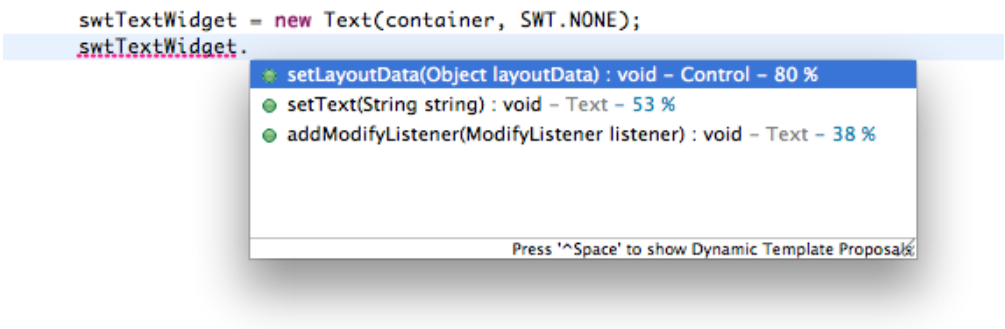

Figure 3.3: The Eclipse auto-completion window with completions sorted by relevance by the Eclipse Code Recommenders plug-in.

The project also provides other auto-completions like snippets which the developer might find useful in the current context. When subclassing a class and wanting to overwrite a method, snippets are provided that contain the most typical use-cases.

The project aims at giving the developer exactly the information he needs when programming. Unfortunately, it is mostly limited to classes which he already knows and toggles the auto-completion on. It cannot recommend utility classes or other classes which fulfill the functionality needed.

Hippie Completion

Hippie Completion is part of the *Codetrails Connect* Eclipse plug-in. It is developed as a commercial extension to the *Eclipse Recommenders Project* by the same developers.

[9]Eclipse Website: `http://www.eclipse.org` (last accessed October 14, 2013)

The difference between the *Eclipse Code Recommenders* and the *Hippie Completion* is the way the proposals are sorted. While the *Eclipse Code Recommenders* uses the number of calls to a method as found in some code, the *Hippie Completion* collects crowd-sourced completion events. Every time a developer, who has the plug-in installed, triggers the auto-completion to complete some piece of code, the choice he makes is sent to a remote server which stores it in an index. When a request is received, the server offers information as to which method of a given class was called the most by other developers.

This provides the benefit of not being limited to the code which the project can analyze, but also takes code written in closed-source projects into account. Additionally, it does not depend on models being rebuilt in the background. The developer can decide for which classes he would like to upload statistics, with the default only being highly utilized open-source projects like the *Apache Commons* or the *Eclipse* APIs.

3.3.3 API Method Recommendation

Heinemann [15] proposed an *Eclipse* plug-in making use of his *Context-Dependent API-Method recommendation* introduced in section 2.8.

The plug-in provides an *Eclipse* view which displays a list of methods matching the programmer's current context (section 2.8), after they have been requested using a keyboard shortcut.

The system is implemented as a local service, which implies that every developer has to enable it and run the initial analysis on his local machine before the service can be used.

Figure 3.4 shows the recommendation screen made available by the plug-in. The documentation for each recommendation is presented in the *Eclipse* Javadoc view, to aid the developer in understanding its functionality and purpose.

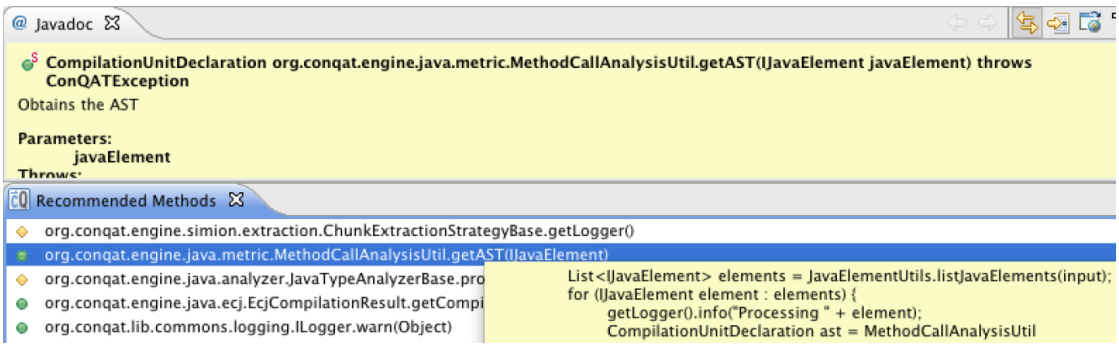

Figure 3.4: API Method Recommendation View displays methods with similar context to the programmer's current context [15].

4 An Incremental Code Search System for Reusable Methods

This chapter covers the design of our own code search system and our implementation of the idea as part of *Teamscale* (section 2.11).

4.1 Scope

To help developers find reusable code, we want to develop a system that can be used to locate code on company internal code repositories. We do not attempt to index large amounts of open-source software from the Internet, as a number of tools are available for this task (section 3.1) and because, in most cases, additional information like code review states or architectures are unavailable for these projects.

We aim at making the entirety of a company's code searchable, specifically not only the code that the developer has checked out locally, but also the code that the developer is not working on and which remains in the company's repositories. This should help developers find reusable code that has been created in other parts of the company and of which he was not yet aware, thus fully utilizing the company's code's potential.

Third party libraries will not be included in our approach, as their source code is usually not readily available as part of the companies' repository. We believe supporting libraries has potential and consider it part of future work (chapter 7).

Our project has a focus on automatically recommending reusable methods, without removing the possibility of manual search. We want to be able to provide results without developer intervention and without the usage of complicated query languages. Instead, search results should be found by extracting the search query from the programmer's current context to find reusable methods. The index used by the back-end should be easily usable for manual searches as well. Manual search is future work, as it is beyond the scope of this thesis.

We aim at making the analysis results available to the developer as quickly as possible after they have been committed. To achieve this goal we want to use an incremental analysis approach, that only updates the database commit by commit and does not have to re-index the entire code base completely, as this would be very time and resource consuming. Using an incremental analysis significantly decreases the time until the developer can retrieve results from the source code's latest revision, as results are made available within minutes or even seconds after committing the changes to the repository.

4.2 Approach

We want to build a system that provides up-to-date data on reusable methods in the current code base and allows the developer to query this data in multiple ways.

As projects with many developers can see many commits per hour or even minute[1], it becomes very important to keep the time between a commit and the latest data becoming available to the search engine as small as possible. We solve this problem by designing the system in a way that it will only analyze the recent changes introduced by the commit and then updates the information present in the back-end. We propose an incremental approach by reading the repository commit by commit, always updating the data when a new commit is pushed to the repository. As the systems can grow very large, it is vital to only work with changed data instead of re-analyzing the entire code-base for every commit in order to keep the time needed for an analysis and update as low as possible.

In order to be useful in an automated environment, such as in an IDE's auto-completion, we need the query time to be sufficiently fast. This can not be achieved by running a live search on the entire data collection for each request. Instead, we have to analyze the data beforehand and store it into an indexing system, which allows us to perform real-time queries on that index. By using the incremental system we can update the index quickly after changes have been made, ensuring the correctness of the information provided to the developers.

We want to allow access to our data to multiple clients at the same time. We propose using a service based approach, which allows any client to send a query to the service and receive the results it needs.

To provide useful information for the developer, we need to define how we find and rank reusable methods. In response to *RQ 1: What are useful metrics and techniques for a code search system?* (section 1.4) we propose a number of metrics and techniques that can be used for this purpose:

- Method Call Contexts (section 2.8)
- Signature Matching (section 2.4)
- Code Review Ratings (section 2.3)
- Architecture Analysis (section 2.9)
- Code Reusability Metrics (section 2.5)

These metrics and the data required to use the techniques need to be calculated and saved to the index together with the method definitions to make them available for search queries. How the metrics are used is explained in section 4.2.2.

We want to provide two distinct clients for our main system design: an automated method recommendation system and a text based code search.

[1] Google has more than 20 changes per minute: `http://google-engtools.blogspot.de/2011/05/welcome-to-google-engineering-tools.html` (last accessed October 14, 2013)

The automated method recommendation system is meant to be part of the IDE's code completion system. Using the current development context at the point the developer triggers the auto-completion, a query is automatically created and sent to the service to retrieve results. These results should then be shown as part of the regular auto-completion to provide seamless integration into the developer's normal development process. No interaction on the developer's part is required, making this client appear like a recommender.

The text based system gives the developer the opportunity to enter keywords, for example a name of a method, into a search field and start a query against the index. The client will then display a list of reuse candidates. It can be extended to contain search fields for the expected return type, the parameter names and types as well as selections about which code review color the developer expects and from which package he would like to call a method, taking the architecture conformance into account. This type of client is a helpful instrument for the developer if he is planning to implement a given feature-set and wants to find out whether or not parts of it are already implemented elsewhere in the system.

These functionalities can be split into multiple components, that make up the system (fig. 4.1):

- An incremental repository reader that reads commits from a repository
- An analysis/data-collection component that is run for every change
- A central index that can be accessed by services
- A code search service that can be queried by clients
- Clients using the service as data source

The *incremental repository reader* reads all changes from the repository commit by commit and triggers the *code search analysis*, which extracts all needed data and stores it into the *central index*. The *automated IDE search* and *text-based search* clients can now query the code search service for results. It uses the *central index* to generate results and return them to the clients (fig. 4.1). All components are explained in further detail in the following sections.

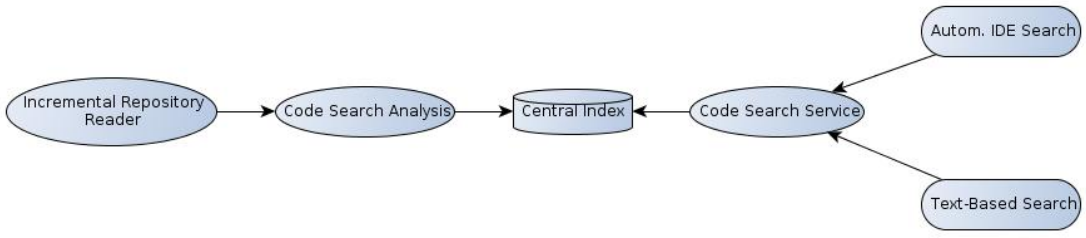

Figure 4.1: Components of our system

4.2.1 Incremental repository reader

In order to build an incremental analysis, we need a system that reads commits from a repository one by one and allows another component to be run after each commit has been read. The component has to be able to access all data collected from the repository in order to collect necessary information or analyze it.

We want to collect the following information:

1. Which files were changed in each commit
2. Commit content (e.g. files)

We need this information to update our back-end with new data. The exact changes determine what part of the source has to be re-analyzed and which part of the index has to be updated for the given commit.

For maximum flexibility we store this information in a historized way, meaning that we can access all the details for each commit at any point during the code search analysis. The historized information is especially important during our evaluation (chapter 5). The information is stored in a central index that can be accessed from other parts of the code or can be used as a back-end for the central search server.

4.2.2 Code Search Analysis

The code search analysis component analyzes the source code and extracts all necessary information to make it easily searchable. All the collected information is stored into the central index.

During the analysis we propose collecting different kinds of information about the code base. Generally we collect all methods found in the code and store additional types of information for each of them. These are explained in the following sections. We have evaluated the usefulness of the proposed ideas in chapter 5.

The collected data will be used by the central index and the services implemented on top of it to generate useful search results for queries.

4.2.3 Method Signatures

Besides storing the method's name for each method we find, we want to store the entire method signature so that signature matching can be performed (as introduced in section 2.4).

Having method signatures in the index allows us to perform many types of queries. Most importantly it can be used for an auto-completion system, as it makes sense to only allow results with a certain expected return type, if the expected type is available from an editor like *Eclipse*. The signature is also useful for manual queries, as the programmer might have a good idea of what he needs in terms of argument and return types and can then use that information to query the service.

This means we have to gather and store the following information for each method:

- Name
- Return type
- Argument types
- Argument names

Additionally we need to identify super- and subtypes, so that we can easily perform generalized and specialized type matching. For best performance these should be stored during the analysis phase, as calculating them at runtime can be very expensive (especially for subtypes, as it requires traversing the entire type tree).

4.2.4 Reusability Valuation

In order to provide good results the system has to evaluate the usefulness and reusability of a method given a certain query. A method judged as highly reusable should be ranked higher than a method that was rated with a low reusability. We will be using three metrics and techniques to evaluate the reusability and usefulness of a method for a given query: context-dependent method recommendation, review ratings and architectural information.

4.2.5 Context-Dependent Method Recommendation

Our main technique to retrieve results, for queries generated by automatic tools such as an IDE auto-completer, is context-dependent method recommendation (see section 2.8). Its usefulness has been demonstrated [15] and it allows us to find methods useful in the programmer's current context.

Listing 4.1: A section the programmer is current working on

```
1 int lower = 5;
2 double minimum = <QUERY>
```

For example given the code in listing 4.1 and the sample context dataset in table 4.1, the query will contain the programmer's current context C for the query position ($<QUERY>$), which is *[minimum, double, 5, lower]* for a *lookback* of 4.

Method Name	Context
MathUtils.calculateMinimum	[lower, minimum, double, math]
MaxUtils.calcMax	[maximum, double, high, calc]
AvgUtilities.average	[double, lower, maximum, avg]

Table 4.1: Example dataset of method names and the associated contexts.

Matching this context C to the three contexts in table 4.1 results in obtaining the best match for *MathUtils.calculateMinimum*, as 3 of 4 words in the programmer's context

could be matched to its context (lower, minimum and double). The two other methods only have matches of 1 of 4 and 2 of 4.

4.2.6 Review Ratings

We believe that review ratings (see section 2.3) are useful information for our code search engine. Code that is still under review is not yet finished, still undergoes frequent change and potentially contains more bugs. Therefore it is less suitable for reuse than already reviewed code, which has been approved and marked as complete. Few actual studies about the effectiveness of reviews for maintainability and fault detection exist, but they indicate a positive effect [32, 12, 13].

Code reviews alone are not a useful method of finding reusable methods given a specific query, but help with refining the results as a secondary rating by emphasizing good results (reviewed) and demoting others (un-reviewed).

4.2.7 Architectural Information

Many software projects are designed based on a software architecture, therefore we believe this information should be considered when creating a code search engine. Our idea is that a result should be ranked higher if it conforms to the system's architecture and lower if it does not, promoting architecture preserving reuse.

Considering the architecture allows us to explicitly forbid access to methods which would break it, if they were to be used. Completely blocking results that break the architecture specification could help keep a high level of architecture conformance throughout the project's development. However, it also introduces the risk of missing good reuse opportunities, which the programmer could have fixed to adhere to the given architecture. Allowing results with a bad conformance value but reducing their overall score should help displaying a full set of results but emphasize more appropriate methods if available. Another valid approach could be to not affect the ranking at all, but display a warning to the programmer if he is reusing a method at a position where the method call will break the system's architecture.

For our prototype we will allow results which will break the architecture, but penalize the result. To our knowledge no research regarding the usefulness of this approach exists. We provide experimental data as a first step to fill this gap in chapter 5.

4.2.8 Central Index

All information that is gathered in the main analysis component is stored into a central index or database. By collecting all data in a central place we can easily create services on top of them to provide functionality for clients.

4.2.9 Code Search Service

The code search service provides an interface for clients to send code search queries and retrieve results. It is responsible for handling all incoming queries, collecting the necessary

data from the central index and possibly enhancing it depending on the requested result type. For example a code completion request might use a different weighting scheme for the return type than a method search using a web interface – while it could be seen as a must-have requirement for the completion service, the developer using the web interface might use it as one of many parameters of the query, therefore making it a less strict requirement.

4.2.10 Clients

To provide useful features for the developer, different sorts of clients can be implemented using the proposed code completion and code search services.

Code Completion Client

The code completion client (e.g. the *Eclipse* auto-completer) suggests methods matching the current programming context. The client takes the context (section 2.8) and available signature information and retrieves matching functions by sending a query to the code search service. The returned results are then displayed to the programmer together with the default (see section 2.1) auto-completion results, fully integrating it with the developer's known development environment without disrupting his normal work-flow.

Method Search Client

The method search service works similar to a classical Internet scale code search service (see section 3.1). The developer can input a possible method name and additional information like the expected parameter types and names. Just as the code completion client, this client can then query results from the code search service that match the developer's query, hopefully providing the developer with a useful result of reusable methods he was searching for.

4.3 Implementation

This section will give a complete overview of our implementation of the proposed system based on *Teamscale*, *Lucene* and *Eclipse*. Our implementation is designed to be easily extendable to work with multiple programming languages, but for this thesis we will focus on Java as our primary target language, as this is the language we are most familiar with and that is best supported by the tools we use.

4.3.1 Architecture

Our system is implemented as a four layer architecture (see fig. 4.2). The three lowest layers are part of *Teamscale*, while the upper layer is made up by external clients like a web interface and *Eclipse*.

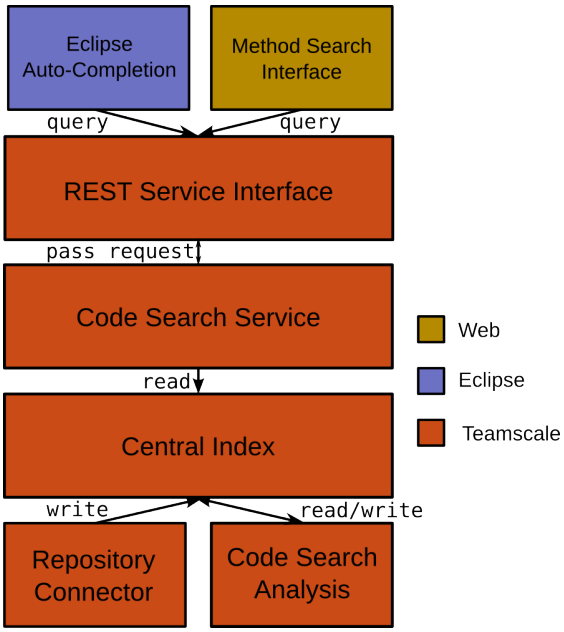

Figure 4.2: Our code search system's basic architecture

Dataflow

The analysis starts by reading commits from a version control system. *Teamscale* provides repository connectors for *Git*, *SVN* and *TFS* and we use them without changes. It reads the necessary data from the repository commit by commit and stores it into the index. By doing so, it triggers additional analyses on the new changes resulting in any analysis' relevant data in the index being updated. Our code search analysis is part of the analyses and collects the necessary information needed by our code search service.

The code search service is not tightly coupled to the analyses, therefore able to serve requests even while new commits are being analyzed. This is possible because it reads

all the necessary data from the index, resulting in full usability of the services during the entire analysis time, albeit with slightly outdated data for short time intervals. As only very small parts of the entire dataset are changed in a single commit, we find this acceptable.

The client can send any *Lucene* query to the code search service, which will then use *Lucene* to parse the query, retrieve results from the index and return them to the client.

4.3.2 Data storage

We use a *Lucene* (introduced in section 2.10) index to store all retrieved information and make the information searchable. This relieves us from the task of implementing our own system for this task. In *Lucene* data is stored in a *Document*. In our system that means we will store every method declaration as one document. For each document we have a set of fields containing the relevant information, needed for signature and context matching. A list of fields is available in table 4.2.

Field Name	**Type**	**Analyzer**	**Example content**
path	String	Keyword	com/test/type.java
method_name	Text	Simple	com.test.Type#aMethod
return_type	Text	Whitespace	java.util.List java.lang.Object
argument_types	Text	Simple	java.lang.String
argument_names	Text	Simple	fileName
number_of_arguments	Int	Keyword	1
static	String	Simple	true
context	Text	Standard	string foo whatatext foo substring
packages	Text	Whitespace	type.java
code_rating	String	Simple	RED
ID	String	Keyword	com/test/type.javacom.test.Type#aMethod1
unique_id	String	Keyword	com.test.Type#aMethod(String)

Table 4.2: The fields that are stored into the *Lucene* index for every document.

Field Details

The following section will explain the details of every field used and listed in table 4.2.

path The path field contains the path to the file in which the method was found. This is useful for the auto-completion to be able to point the programmer to the correct file in case it is not checked out locally.

method_name The method_name field contains the method's name. It is stored as text, to make it easily searchable using full text search. It is stored together with the surrounding class's fully qualified name, to make the name unique.

return_type The return type field contains the fully qualified name of the return type and all its supertypes. The supertypes are stored here to make sure this field also matches when requesting a supertype of the original return type.

argument_types The argument_types field contains the fully qualified name of the argument types present in this method. We store them all in one field, as the ordering does not play any important role for reusability. This field could be extended by the subclasses of all these types, allowing for signature matching for more specialized types. We did not implement subclasses here as it is out of the scope of the thesis.

argument_names The argument_names field contains the argument names used in the method's signature. These can be useful for full-text search later.

number_of_arguments This field contains the method's number of arguments. This is used only in the shallow parser to make a rough mapping of method calls to method definitions possible, as no type information is available. This field is not used in the AST based analysis. It is the only number based field in our document and requires workarounds in *Lucene* to be updated properly[2].

static Contains "true" or "false" indicating whether or not the method is static or not.

context This field contains all words from all contexts (sections 2.8 and 4.2.5) found for the given method. The construction of this string is detailed in section 4.3.4. This is the only field we use the *StandardAnalyzer* on to utilize the full power of *Lucene*'s full text matching capabilities.

packages This field contains the fully qualified name of all packages from which this method can be called according to the architecture. This field is used to implement architecture conformance for results. The query can contain the package from which the method is being called, matching this field if it is allowed to call the method from the given package.

code_rating This field contains the code rating, which is one of *RED, YELLOW* or *GREEN* according to the *LEvD* process (section 2.3.1). This field is used to request which review states are accepted in a query.

ID The field *ID* is used to store a unique id for each method, as created by the analysis using the shallow parser. It is made up of the file name, the method name and the number of arguments. This is not completely unique, but is the best we can do with the shallow parser framework in order to identify a method for later changes.

[2]Our problem and solution on stackoverflow.com: `http://stackoverflow.com/questions/17109371/lucene-search-query-using-a-intfield-not-working-after-document-update` (last accessed October 14, 2013)

unique_id In the context of the JDT we have the possibility to create a fully unique id, which we store in *unique_id*. This is made up of the fully qualified class name and the methodname + argument types.

4.3.3 Code Search Analysis

This section will explain the details of how we implemented our code search analysis. Our analysis takes a set of source files as input, extracts all information to create new documents for each method declaration and stores them to the index with the goal of making the information available for clients. There are two different approaches for the implementation: first we implemented it using a shallow parser (see section 2.6). Secondly, we used a more language specific approach using the Java abstract syntax tree. Each analysis is implemented as a set of *Teamscale* triggers (see section 2.11), in order to run in *Teamscale*'s incremental analysis engine.

Shallow Parser based Analysis

As shallow parsers are faster in reading source code and creating a model for it than compilers, we attempt to implement parts of our system using a shallow parser as a first step. Because we want to provide an up-to-date index as quickly as possible we highly value speed when extracting information from the source code.

The shallow parser framework provided by *Teamscale* supports many languages, but we will focus on Java for our first implementation.

As a first step in our analysis we want to find all method declarations. Using the shallow parser to retrieve the data necessary to fill the document, as described in section 4.3.2, works for method definitions that do not use generics. We can retrieve the method name, argument types and names, as well as the return type in many cases. Because the shallow parser does not provide this information entirely by default, we have to work on the token stream of the parsed source code to retrieve it. This makes a truly language independent implementation difficult, but through use of abstraction we believe that it is possible to obtain a framework that can be extended to new languages.

Methods using generics are difficult to handle using the shallow parser framework, as method signatures vary in their look considerable. In listing 4.2 we list a number of signatures that might pose problems. Using simple whitespace splitting or counting the token offset from the method name will not work for these examples. Instead it is necessary to count closing and opening braces and generally use a lot of programming language specific knowledge. The offset problem can be solved using abstraction and small language specific functions, but does require a noticeable effort and deep knowledge of the language's syntax.

Listing 4.2: Different Java signature examples posing problems for shallow parser

```
1 private final static String[] getType(List<IToken> subList, int index)
2 private static final List<String> getType(List<IToken> subList, int index)
3 public abstract List<Map<String, int>> getType(List<IToken> subList, int index);
4 public abstract List<Map<A, B>> getType(List<A> subList, int index);
```

For simple types it is often possible to determine the actual return type of a method, but for classes this can be unclear. For example *List* could be *java.util.List*, but it could be any other class called *List* (e.g. *java.awt.List*) as well. This is especially problematic if generics like in line 4 in listing 4.2 are used. It is not possible to tell which type is actually expected from reading the identifiers *A* or *B*.

This problem could be solved using rather simple heuristics utilizing language features like imports. Doing so would not line up with our goal of a language agnostic implementation. In the long term it would not provide any benefit compared to using a full abstract syntax tree generated by a compiler, which can provide the necessary information.

The second step of the analysis is extracting contexts from the source code. This is only possible for static method calls using the *ClassName.staticMethodName()* syntax.

The shallow parser does not provide any support for finding method calls, as the smallest entity it knows of is a statement. A single statement can contain multiple method calls or none at all. Finding method calls can only be done by searching through the list of tokens contained in the statement and trying to find method call patterns like a dot, followed by text, followed by an opening brace, followed by some code, followed by a closing brace. This detects many method calls, but we cannot easily identify on which class the method is being called. For static methods using the specified syntax, we can find the class name, which can be unique and allows us to map the context correctly. Again this problem could be fixed by scanning the entire file for variable declarations, keeping an index of them and using the imports to find out exactly to which package a class belongs, but this would require a large development effort and is very language specific.

We can not provide sub and super-types for the return types and parameter types, as a type hierarchy is very difficult to obtain using only shallow parser technology and not a real parser.

Using the shallow parser leaves us with a simple method declaration indexing, that works well for methods without generics in many cases. This can be combined with context collection for static methods, resulting in a usable system for static method search. The problems explained make a clear case for implementing a language specific version of the code search analysis using a compiler that produces a complete abstract syntax tree, which provides all the necessary information. This will be our second step.

RQ 3 - Can a shallow parser replace a full parser in code search engines? During the implementation we found that a shallow parser cannot completely replace a full parser for our purpose, unless a great effort is made in expanding the shallow parser system to contain most of the information provided by a full parser.

The biggest challenge we experienced is mapping method calls to already indexed methods in order to store the surrounding context identifiers. This is because we cannot determine the argument types used and the instance type on which the method is called. Because of the lacking type information we cannot clearly map the method call to a method in the index.

The shallow parser can however be used for parts of the system, namely the method declaration indexing with a few compromises such as no super/sub-type resolution.

Abstract Syntax Tree based Analysis

As using the shallow parser framework proved limiting for our purpose, we decided to use a full abstract syntax tree (AST) as provided by the *Eclipse ECJ and JDT* frameworks. This approach is only usable for *Java* and the developed code can not easily be reused for other languages. The *Eclipse ecj* compiler provides facilities to traverse the AST using the visitor pattern [3]. The visitor class *ASTVisitor* provides callbacks for method declarations and method calls which we will use to collect the data we need.

We can extract a signature from every method declaration using the information provided by the compiler. It allows us to query the return and parameter types for each method declaration and we can collect the supertypes for a given return type using the compiler's type hierarchy.

Collecting supertypes is interesting for the signature matching process, as when a method returns a given type X, it is also usable at any given position where any supertype of X is required. For example in Java, if *java.util.ArrayList* is returned, it can also be used to assign the results to a normal *java.util.List*, which is a supertype of *ArrayList*.

We decided not to collect subtypes, even though this could be interesting for signature matching with respect to parameter types. For parameter types the opposite of the return type is true, meaning if the method takes a type Y as parameter, all subtypes of Y are valid as well. We decided against collecting sub types as this is more difficult to implement and we do not have a use case for the parameter type matching in this thesis.

The availability of the *MessageSend* signal makes finding method calls a trivial task. In contrast to the shallow parser, the compiler provides all information needed to create a unique signature (method name, return type and parameter types in Java) and thus makes proper updating of the index possible.

4.3.4 Context Retrieval and Storage

In contrast to all other metrics we store, the data needed for the *Context-Dependant Method Recommendation* can be seen as a global metric (see section 2.11.1). This means that if the context changes, this affects files (or more precisely method declarations in our case) which have not been changed in this commit.

For every method call a context is extracted (see section 2.8). If a file changes, the context around multiple method calls can change and thus affect already stored contexts for methods which have not been added in the same change. As a result we have to be able to update the context for every method at any commit. We do this by storing the context for every method call in a separate table, where we map the filename and method call to the extracted single context. This allows us to remove single contexts when a file is removed or added and change entries every commit.

At the end of the analysis, we combine all contexts for each method to one big context and store it into the *context* field of the *Lucene* document.

[3] Visitor Pattern explained: `http://en.wikipedia.org/wiki/Visitor_pattern` (last accessed October 14, 2013)

By using this approach we avoid re-extracting all contexts over and over for each commit, at the cost of having to store the context twice: once as many single entrys in the mappings store and once as the combined string in each *Lucene* document, for each method in the index.

4.3.5 Code Search Service

The code search service is a basic web service that takes a set of parameters. It can be queried by sending an HTTP *GET* request containing the following parameters:

- Lucene query (see listing 4.3 for an example query)
- Number of desired results

The service is only a very thin wrapper around *Lucene* at this point and just passes the query to *Lucene* without any changes. All results returned by *Lucene* are returned to the caller as a *JSON*[4] string, containing all available information (see section 4.3.2) about any method that is being recommended.

4.3.6 Eclipse Auto-Completion Plugin

We provide a working prototype of an *Eclipse* plugin, which includes a new auto-completion provider, that connects to our code search service. It collects the current context (as defined in section 2.8), the expected return type at the cursor's position and the current package and forms a simple *Lucene* query, which is then sent to the service (see example in listing 4.3).

Listing 4.3: Lucene Query Example

```
1 return_type:'com.jgoodies.forms.builder.DefaultFormBuilder' OR
2    packages:'net.sf.jabref.oo' OR
3     context:'b getpanel setborder borderfactory createemptyborder b getpanel' AND
4     (
5       code_rating:'GREEN' OR
6       code_rating:'YELLOW'
7     )
```

The returned results are displayed as auto-completion results and are intended to be used in exactly the same way as the default auto-completion entries. An example result set is shown in fig. 4.3.

[4] JSON website: `http://www.json.org/` (last accessed October 14, 2013)

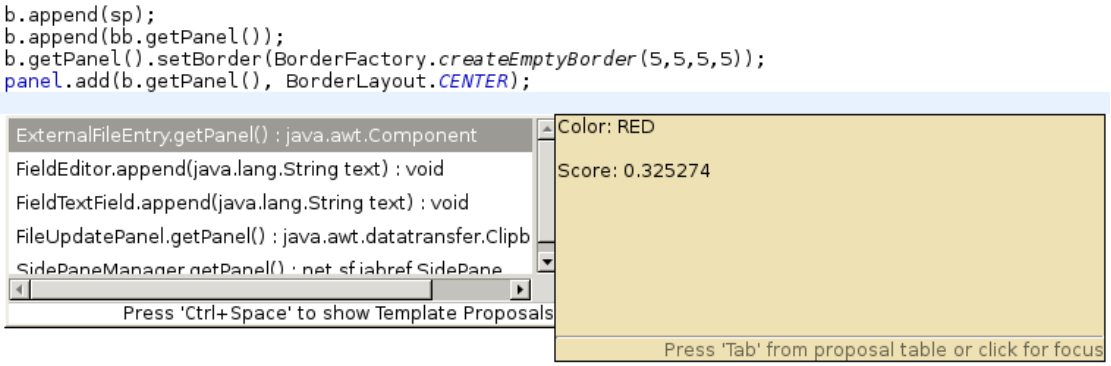

Figure 4.3: Demonstration of our *Eclipse* auto-completion.

5 Evaluation

This chapter covers the details of the evaluation performed to measure the usefulness of our approach.

5.1 Evaluation Design

In this thesis, we propose an incremental evaluation approach that closely follows the stuy object's actual development history.

5.1.1 Approach

Our idea is to leverage the experience we gained by developing an incremental code search engine to design an evaluation system based on a similar approach.

In every commit a certain number of changes are performed by the programmer, including adding new method calls. For each newly added method call we pretend the developer used our auto-completion tool to trigger a query to our system at the position where it was added. We are interested in seeing in how many cases the code search system returns the programmer's chosen result in its top M results. If the code search systems returns the result matching the programmer's choice in the top M results, we consider the result to be correct.

Because of the incremental approach we can "travel through time" and measure the accuracy for every commit based on the back-end data available at that point in time. This means we get to evaluate our system using a study object automatically over a given timespan, without being required to have collected any additional data besides the study object's code history during development. This is in contrast to most common evaluation approaches like ten-fold cross-validation, which only work on one single snapshot of data.

We define the following variables:

- *hits = number of times the programmer's choice appears in the top M results*
- *queries = number of queries performed*

From these variables we derive our evaluation metric: the *Hit/Query Ratio*

$$hit\ ratio = \frac{hits}{queries}$$

The *Hit/Query Ratio* tells us how often our system would have proposed the programmer's choice in the top M results. This metric provides insight on how reasonable the

recommendations made by our system are. We want the *hitratio* to be as close as possible to 1, as a higher value indicates more matching results for the number of queries.

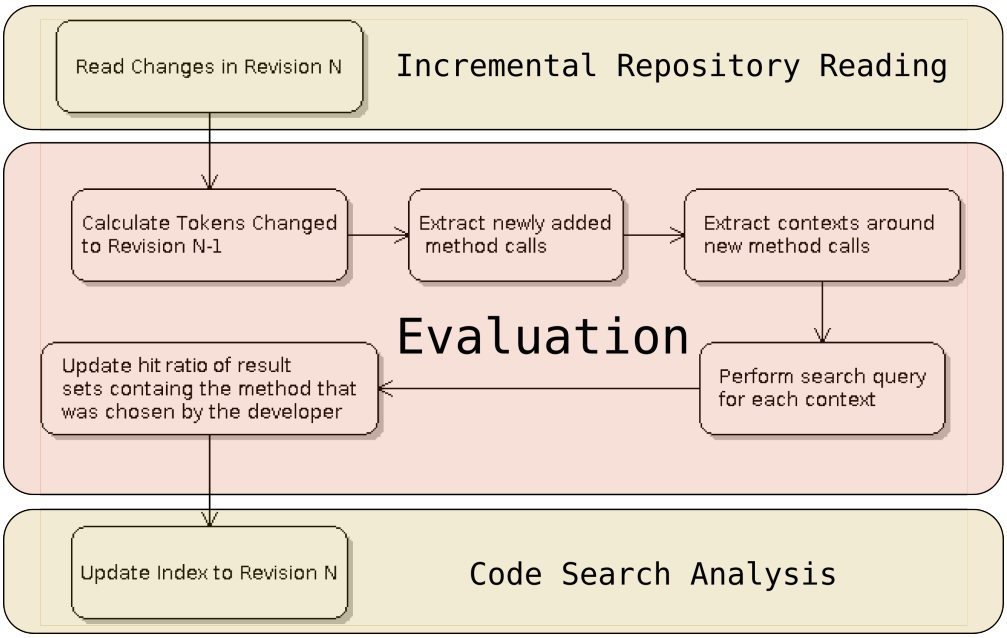

Figure 5.1: The evaluation steps performed for each revision of the study object.

Figure 5.1 shows the single steps needed to perform the evaluation. After the changes for a given revision N have been read they are passed to the evaluation. There we first have to calculate the actual changes made in comparison with the previous revision N-1. This is necessary to extract only the newly added method calls and the contexts around them. After the methods and contexts have been extracted, we formulate a query for each position a method call has been found. For each result set we check if the method chosen by the programmer is contained in it. By repeating this process for each commit we can incrementally calculate the overall hit ratio of result sets containing the correct choice. After the system has finished the evaluation the code search analysis is started and updates the index to contain the data added by revision N.

We evaluate different systems each with different sets of metrics used for the code search system. We can disable or enable the following parameters:

1. Expected Return Value
2. Code Rating
3. Architecture Analysis

The parameters have been chosen because these are the parameters we can fill using the auto-completion in the IDE. We can derive the required return value (where appropriate) and set a code rating we wish our result to have (in our case: YELLOW and GREEN).

By passing the package, the programmer is currently editing a file in, we can fill the architecture analysis field.

This leaves us with 8 sets of combinations per study object, if all parameters can be used. For each combination we will calculate the *hit ratio* for different result sets:

- 1-5, 10 and 20 results returned by our code search system for each combination of settings
- The 1-5, 10 and 20 most used methods in the system

Result sets of size 1-5, 10 and 20 results are evaluated to answer the question of how many results we should display to the programmer for the best *hit ratio*. We expect larger result sets to result in a better *hit ratio* but it is uncertain of how much better the result will be. The more results we have to display, the more entries the developer has to go through to find a result. If too many results are displayed and no correct proposal is provided developers will quickly turn away from the tool [3].

We use the result sets containing the 1,5, 10 and 20 most used methods in the study object as a baseline for comparison to the results from our code search system. We do this to make sure it produces something more useful than the most obvious metric - the number of uses of a method.

5.1.2 Study Objects

We evaluate our code search system using two study objects – *JabRef* (section 5.1.2) and *ConQAT* (section 5.1.2) – as base to gain more insight as to which parameters are useful. An overview of basic information about the study objects is given in table 5.1.

Name	Lines of Code	Number of Commits Analyzed	Timespan Analyzed
JabRef	~150.000	3815	$9\frac{1}{2}$ Years
ConQAT	~340.000	5275	1 Year 2 Months

Table 5.1: Basic facts about *JabRef* and *ConQAT*, the study objects used as base for our analysis.

JabRef

JabRef is the first system we use for the evaluation. "*JabRef*[1] *is an open source bibliography reference manager.*"

JabRef was not developed using the *LEvD* (section 2.3.1) review process. Consequently, no code rating colors are available from the source. We can therefore not evaluate this metric on this project and can only compare using expected return types and architecture analysis information.

In order to create an evaluation using *JabRef*, we derived an architecture for JabRef from its code. This was done by creating a dependency graph between all components

[1] JabRef website: `http://jabref.sourceforge.net/` (last accessed October 14, 2013)

using *Teamscale* and then creating a basic architecture from that information using the *ConQAT* architecture editor. The result is presented in fig. 5.2.

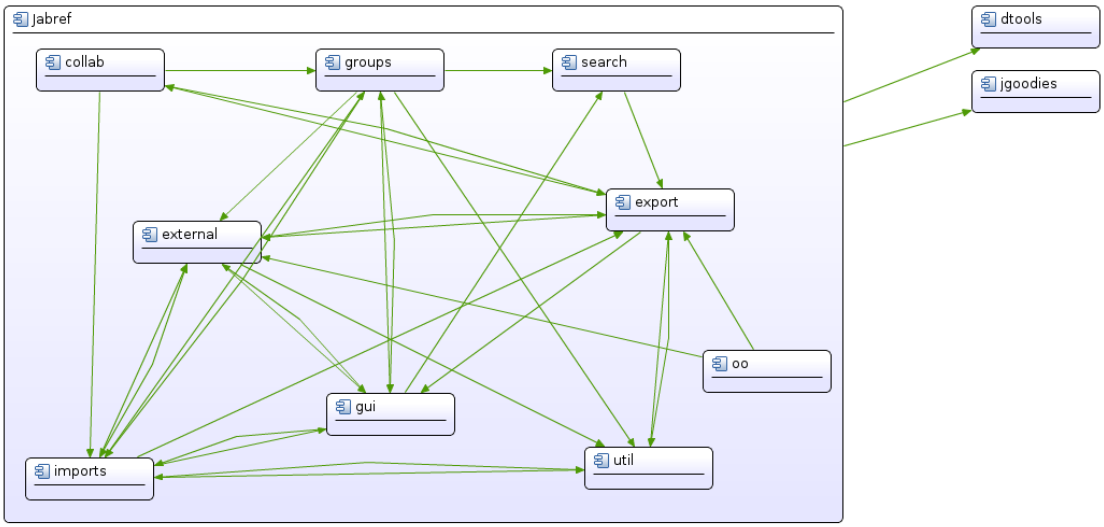

Figure 5.2: Our idea of the *JabRef* architecture.

We analyzed the *JabRef* GIT repository[2] starting at the initial commit with revision '9547faac89' (Dated: Tue Oct 14, 2003). The last commit we analyzed was the commit with revision 'bf47b353e9' (Dated: May 28, 2013). This means we are evaluating on about $9\frac{1}{2}$ years worth of development history totaling 3815 commits over the entire time span. *JabRef* has around 150,000 lines of code.

ConQAT

Our second test system is *ConQAT*. "*ConQAT is an integrated toolkit for creating quality dashboards that allow to continuously monitor quality characteristics of software systems.*" [3]. It can provide an architecture and review ratings and is a system that we can use to fully compare all combinations of our metrics.

We have analyzed all commits from revision 41125 (Date: Sat Jul 21, 2012) to 46400 (Dated: Mon Sep 23, 2013), a total of 5275 commits. The analyzed part of the system contains roughly 340,000 lines of code and spans over a time of $1\frac{1}{2}$ years.

Figure 5.3 shows the part of the *ConQAT* architecture that we use for our analysis. The *ConQAT* architecture was created some of the project's central developers.

[2] JabRef GIT repository hosted at: `http://sourceforge.net/p/jabref/code/` (last accessed October 14, 2013)
[3] ConQAT website: `http://conqat.cqse.eu` (last accessed October 14, 2013)

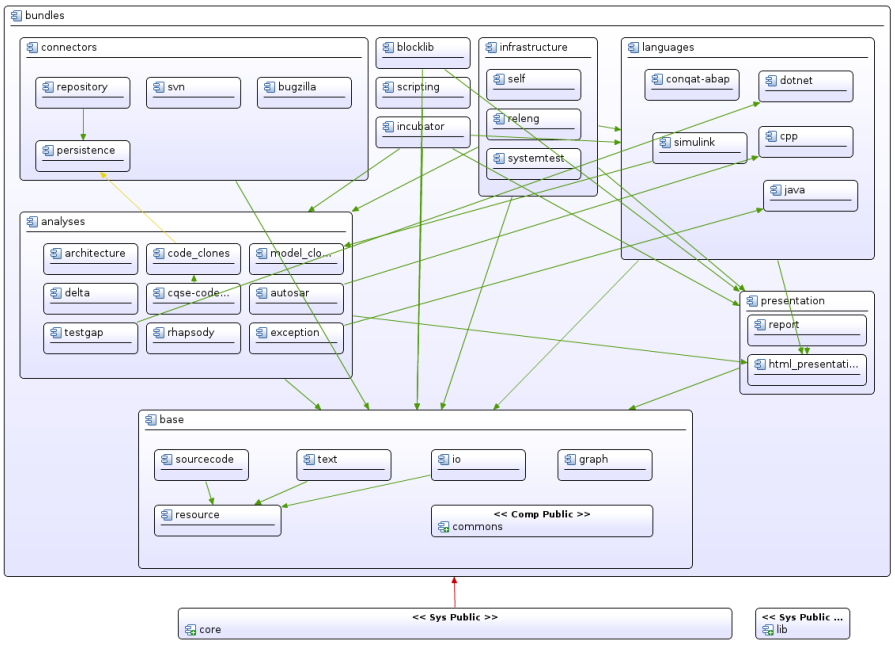

Figure 5.3: Part of the complete *ConQAT* architecture that is used for the analysis.

5.1.3 Implementation

Following our choice of *Teamscale* for the code search system, we decided to use *Teamscale* for the evaluation as well. This provides many benefits like being able to access its internal indexes directly for increased performance.

Most importantly though, it allows us to easily integrate the evaluation into the analysis chain we built for our code search system. This is important, as the evaluation needs access to parts of the current commit's data (e.g. the newest abstract syntax tree to extract the method calls) but also has to be able to send queries against the previous commit's code search base. This can easily be achieved if the evaluation is run just before the code search analysis. This allows us to access any new information we need to create the queries for the evaluation and guarantees the evaluation queries the code search data available at that time, just as the programmer would have done.

Figure 5.4 shows a schema of the *Teamscale* trigger we created. It contains three main parts. First, all changed files are read from the repository. Their content is then extracted and passed to the compiler, where they are compiled and the abstract syntax tree (AST) is created. The AST is passed to the evaluation where the exact changes are inspected and each new method call is evaluated. Besides requiring the changed files, it also has to work with the code's history in order to calculate exactly which tokens were changed in the source code in the current commit. With this information we know which method calls were actually added in the current commit. This allows us to ignore method calls that point to methods which were added in the same commit, as the code search system can not possibly know about them.

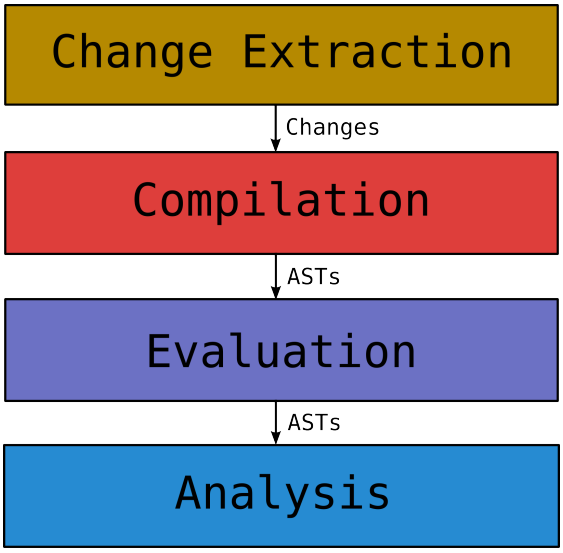

Figure 5.4: A schematic of our trigger and its data flow.

After the evaluation is complete, the *AST* is passed to the analysis, where all changed data is analyzed and the index is updated.

We do not evaluate method calls to methods that have been added in the same commit. Theoretically we would hope that our code search system would present alternatives to the newly implemented methods, thereby leading to more reuse. As we have no way of automatically evaluating if the proposed methods would have been alternatives to the newly implemented method, we do not consider these method calls in our evaluation.

5.2 Results

In this section we present our collected results for each study object.

Results based on JabRef

We present the results based on analyzing *JabRef* in tables 5.2 to 5.5. Figure 5.5 gives an overview of the obtained results.

The graph shows the size of the retrieved result set on the x-axis and the *hit ratio* on the y-axis. Each curve shows the hit ratio development of one combination of metrics over different result set sizes. Each point marked on a curve presents one hit ratio measurement for the marked result set size.

For each setting, the best result has been marked bold in the corresponding table. Clearly the best result combination of parameters for this project according to our evaluation is using return types without the architecture analysis. Overall we get the best result when retrieving 20 results, leading to a *hit ratio* of *0.6566*. The fact that retrieving 20 results leads to a higher *hit ratio* than 10 or even only 5 or 1 result is not very surprising as it

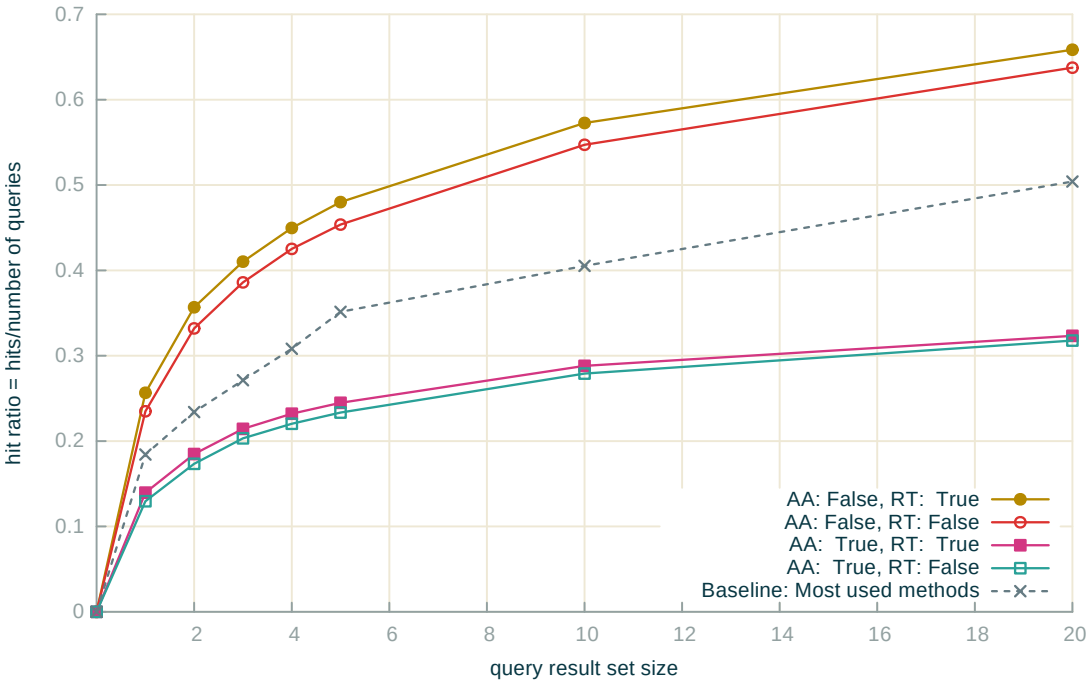

Figure 5.5: Comparison of results based on *JabRef*. (*AA*: Architecture Analysis, *RT*: Return Types)

greatly increases the chances of returning the correct method. The difference between retrieving 5 or 10 results is noticeable with a *hit ratio* increase of 0.08 when using 10, a 16.8% increase. The difference between 10 and 20 is 0.07 or a 11.9% increase. The benefit of a growing result set becomes less with each increase.

Interestingly using the top 20 most used methods every time leads to a *hit ratio* of 0.5041, which means that more than every second method call is to one of the twenty most used methods in the program. This indicates the heavy reuse of a few methods at a very high rate. Using our tool still results in a *hit ratio* improvement of 0.1525 or a 30.2% improvement.

The biggest noticeable difference is between analyses using the architecture analysis and those not using it. Evaluations not using architecture analysis score significantly better than those that do, with result being almost twice as good. This could have a number of reasons: First, the architecture we created is artificial. As we are no *JabRef* developers, we had to extract the architecture from the source code's effective architecture, which means very few restrictions for most parts of the code. Another problem is, that we only have a snapshot architecture for one point in time (now) – *JabRef*'s architecture might have evolved over the past ten years – thus our architecture might not always accurately demonstrate the system's state at that time.

Return Types	Arch. Analysis	Hit Ratio	Total Queries
true	true	0.2229	18419
true	*false*	**0.2997**	*18419*
false	true	0.2105	18419
false	false	0.2814	18419

Table 5.2: 1 result per query for *JabRef*

Return Types	Arch. Analysis	Hit Ratio	Total Queries
true	true	0.3679	18419
true	*false*	**0.5021**	*18419*
false	true	0.3549	18419
false	false	0.4801	18419

Table 5.3: 5 results per query for *JabRef*

Return Types	Arch. Analysis	Hit Ratio	Total Queries
true	true	0.4252	18419
true	*false*	**0.5867**	*18419*
false	true	0.4153	18419
false	false	0.5637	18419

Table 5.4: 10 results per query for *JabRef*

Return Types	Arch. Analysis	Hit Ratio	Total Queries
true	true	0.4722	18419
true	*false*	**0.6566**	*18419*
false	true	0.4665	18419
false	false	0.6387	18419

Table 5.5: 20 results per query for *JabRef*

Results based on ConQAT

We present the results based on analyzing *ConQAT* in tables 5.6 to 5.9. Figures 5.6 and 5.7 give an overview of the obtained results. Similar to the results for *JabRef* the graphs show the size of the retrieved result set on the x-axis and the *hit ratio* on the y-axis. Each curve shows the hit ratio development of one combination of metrics over different result set sizes. Each point marked on a curve presents one hit ratio measurement for the marked result set size. The measurements have been divided on two different graphs for better readability. Figure 5.6 contains all results using parameter combinations not using architecture analysis, whereas fig. 5.7 shows all results using combinations that do use architecture analysis.

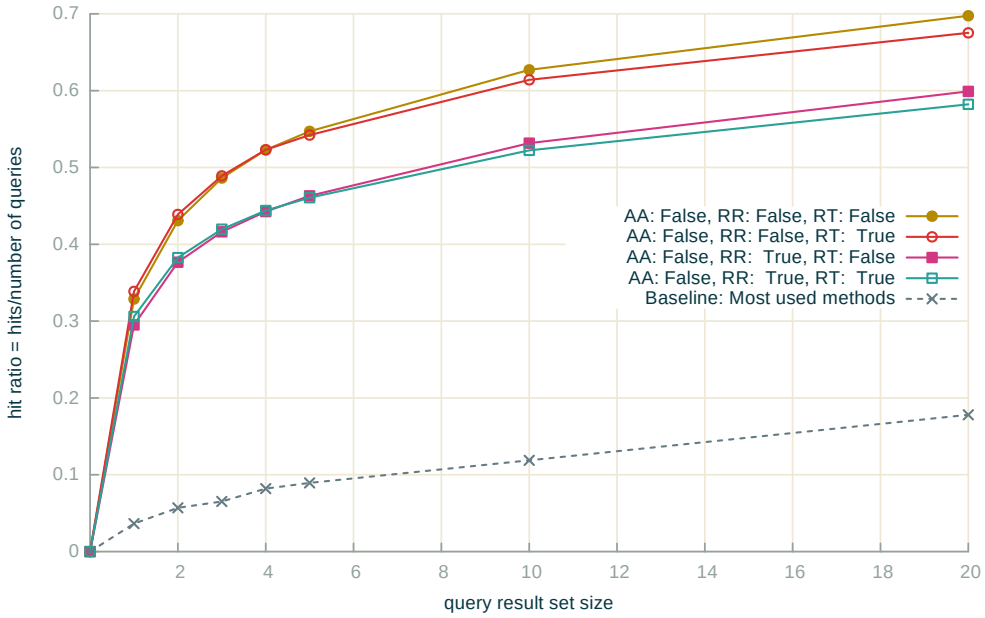

Figure 5.6: Comparison of results based on *ConQAT* without using architecture analysis. (*AA*: Architecture Analysis, *RR*: Review Ratings, *RT*: Return Types)

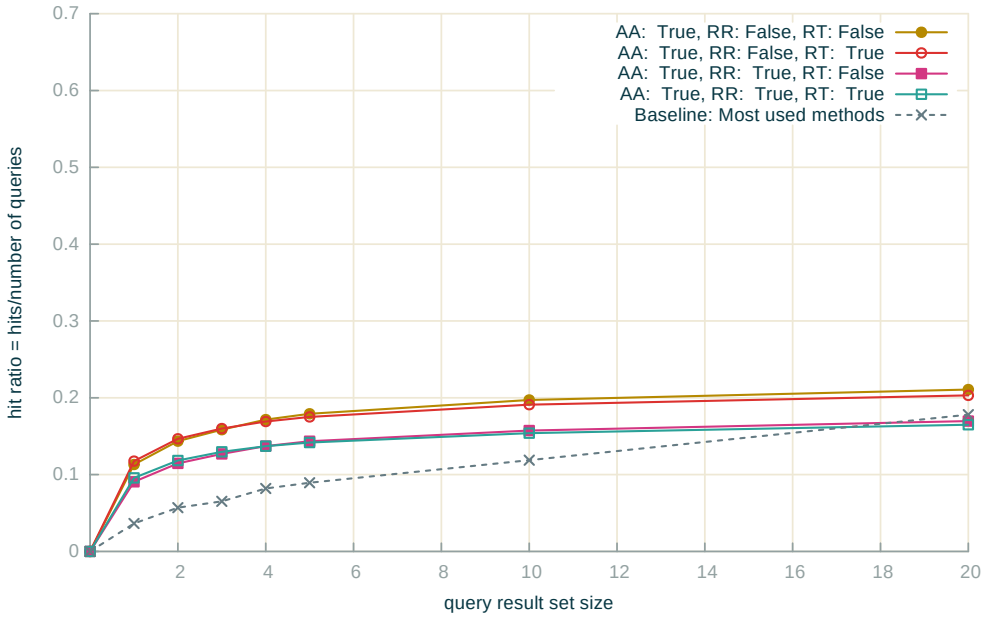

Figure 5.7: Comparison of results based on *ConQAT* using architecture analysis. (*AA*: Architecture Analysis, *RR*: Review Ratings, *RT*: Return Types)

The evaluation using *ConQAT* shows three very interesting results which we will discuss:

- Not using architecture analysis leads to a much better hit ratio than using it.
- The best performing settings are different for 1-4 and 5, 10, 20 retrieved results.
- The baseline result sets containing the N most used methods deliver a very low hit ratio.

Enabling architecture analysis to find better search results leads to a significant drop of the *hit ratio* value (compare figs. 5.6 and 5.7). The best *hit ratio* value with architecture analysis is only about $\frac{1}{3}$ of the best *hit ratio* value that does not use architecture analysis. This is quite surprising as we would have expected it to perform at least similarly for both cases. As with *JabRef* we only have one snapshot of the architecture, possibly leading to inaccurate results due to a wrong architecture in early revisions.

Figures 5.6 and 5.7 show an interesting effect between 4 and 5 retrieved results: The analyses using return types result in a slightly lower *hit ratio* than those not using them. The graphs demonstrate that the settings used in an actual production system should be adjusted depending on how many results are to be retrieved. For example for our auto-completion scenario, we should be using return types only for 1-4 results. For more results we obtain better results when not using them. All other techniques should be turned off. For a web search, using 20 results could be realistic. In that scenario we should therefore disable all extra information to get the best results.

Returning our baseline, the result sets containing the N most used methods in the system, instead of using our code search system leads to very low results for *ConQAT*. This is expected, as *ConQAT* is a large system and the part we analyzed contains over 340,000 lines of code. Since *ConQAT* is a complex tool to create software quality analyses it contains numerous methods for different purposes and does not have one or two central methods that are called frequently. This result gives us confidence that our system scales well with larger systems and that it is vastly better than just using the easiest case, namely the most used methods.

The usage of review ratings seems to have a negative effect on the *hit ratio* value. One reason could be that developers use many unreviewed methods during development. As we evaluate every commit on the repository, we get the entire spectrum of the review chain. This means many commits contain recently created, unreviewed code or changes to such code. Using review ratings as part of the query, i.e. giving extra weight to reviewed methods, will lead to worse results on these types of commits.

Return Types	Arch. Analysis	Review Ratings	Hit Ratio	Total Queries
true	true	true	0.0958	7884
true	true	false	0.1176	7884
true	false	true	0.3061	7884
true	*false*	*false*	**0.3387**	*7884*
false	true	true	0.0907	7884
false	true	false	0.1131	7884
false	false	true	0.2950	7884
false	false	false	0.3288	7884

Table 5.6: 1 results per query for *ConQAT*

Return Types	Arch. Analysis	Review Ratings	Hit Ratio	Total Queries
true	true	true	0.1418	7884
true	true	false	0.1752	7884
true	false	true	0.4607	7884
true	false	false	0.5422	7884
false	true	true	0.1435	7884
false	true	false	0.1792	7884
false	false	true	0.4631	7884
false	*false*	*false*	**0.5471**	*7884*

Table 5.7: 5 results per query for *ConQAT*

Return Types	Arch. Analysis	Review Ratings	Hit Ratio	Total Queries
true	true	true	0.1540	7884
true	true	false	0.1910	7884
true	false	true	0.5223	7884
true	false	false	0.6140	7884
false	true	true	0.1574	7884
false	true	false	0.1971	7884
false	false	true	0.5317	7884
false	*false*	*false*	**0.6271**	*7884*

Table 5.8: 10 results per query for *ConQAT*

Return Types	Arch. Analysis	Review Ratings	Hit Ratio	Total Queries
true	true	true	0.1649	7884
true	true	false	0.2032	7884
true	false	true	0.5822	7884
true	false	false	0.6753	7884
false	true	true	0.1697	7884
false	true	false	0.2107	7884
false	false	true	0.5991	7884
false	*false*	*false*	**0.6975**	*7884*

Table 5.9: 20 results per query for *ConQAT*

5.3 Discussion

We have evaluated our code search system based on two study objects: *JabRef* and *ConQAT*. The study objects vary in size, age and development process. *JabRef* is an open-source project that is developed entirely community based, whereas *ConQAT* is an open-source project that is developed mainly be a single closed group of IT developers. *ConQAT* is more than twice as big as *JabRef* in terms of lines of code.

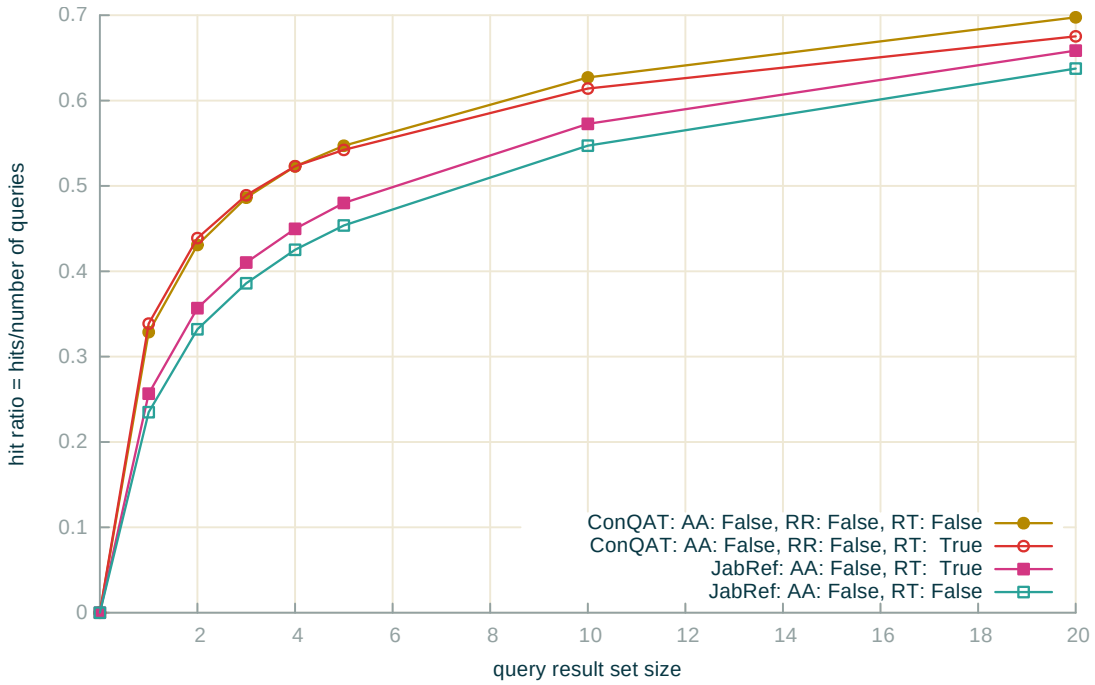

Figure 5.8: Comparison of the two best results for our study objects *ConQAT* and *JabRef*. (*AA*: Architecture Analysis, *RR*: Review Ratings, *RT*: Return Types)

In fig. 5.8 the two best results for *ConQAT* and *JabRef* are presented in one graph. The results are fairly similar for both study objects, indicating that our code search system performs similarly on different study objects. In contrast to *ConQAT*'s results, *JabRef*'s hit ratio lines do not cross at a result set size of four. The gathered data is inconclusive as to whether any conclusions can be drawn from this effect or not.

For *ConQAT* moving from 10 to 20 results we gain an improvement of 11.2%, while using twice as many results. For *JabRef* we gain 11.9%, which is a little more. For *ConQAT* we gain a 14.6% improvement moving from 5 to 10 retrieved results. Using *JabRef* we gain 16.9%. The differences are too insignificant to draw any general conclusions from this. More evaluations would have to be done to generate more conclusive data.

Figure 5.9 displays the *hit ratio* improvement per result for the best two results of *JabRef* and *ConQAT*. Clearly the slope flattens the more results are retrieved, indicating that no large gains in *hit ratio* are to be won by retrieving more results. Judging by the results presented in the graph, we think using 5 results as the default setting for the auto-recommender would be ideal as it leads to a good *hit ratio* and hit ratio improvement per result is still acceptable.

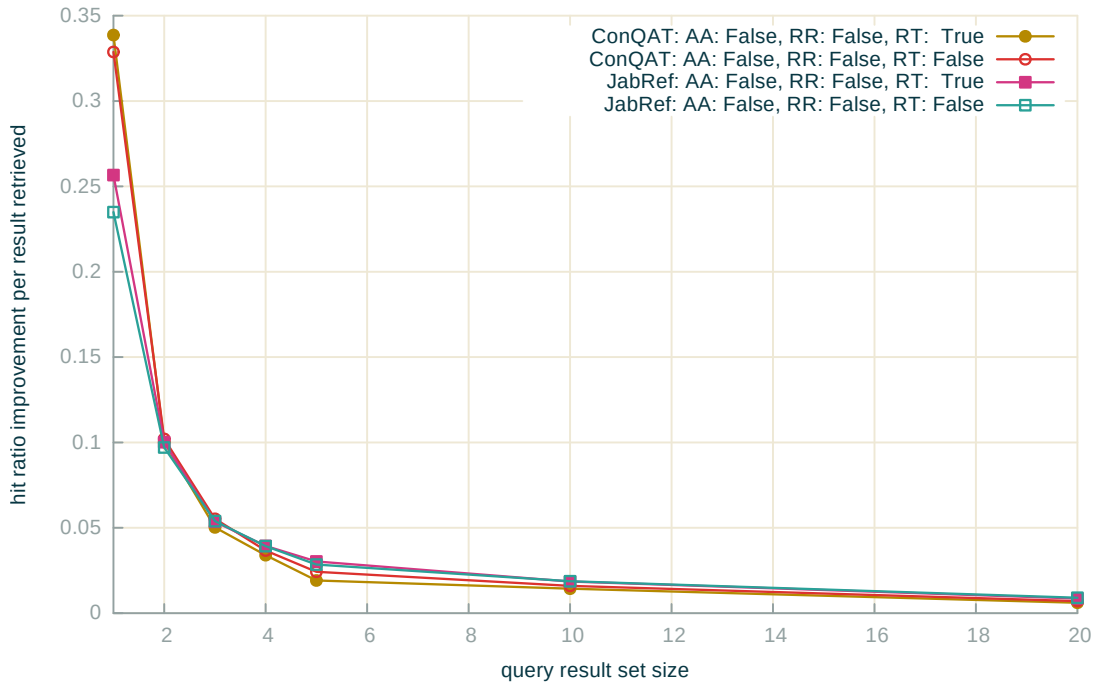

Figure 5.9: Ratio improvement per extra result retrieved in each query. (*AA*: Architecture Analysis, *RR*: Review Ratings, *RT*: Return Types)

We introduced a number of techniques and metrics to improve the search results on top of using the programmer's context: *Signature Matching* (section 2.4), *Architecture Analysis* (section 2.9) and *Review Ratings* (section 2.3).

Using signature matching information seems to work well, as the results for using only

return types are generally the best. For *JabRef* this seems to scale well, whereas for *ConQAT* there seems to be an upper limit of number of retrieved results up to which this is still useful. More study objects would be needed for evaluation to accurately make a judgment whether this technique is an improvement in general or only in certain cases. It certainly is not bad and provided better results for lower numbers of retrieved results (1-10) on both study objects.

The results for architecture analysis are generally a lot worse than those that do not use it. There are multiple possible reasons, as to why the results probably are not better with architecture analysis than without:

- Inaccurate architectures are being used
- Only one architecture version is available for each study object.
- Architecture analysis is not useful to improve the results

The first point is especially true for *JabRef*, as we built the architecture for *JabRef* ourselves, without advice from any of the project's developers. As there was no predefined architecture available, we assume that the project has been developed without a defined architecture, which explains the lack of usefulness of our architecture approach. The architecture used for *ConQAT* should be fairly accurate for the current development state, as it was developed by two of the project's most senior developers. Unfortunately this does not seem to help with retrieving better results.

Both architectures have the general problem of only being available for the current state of development. Since we are analyzing thousands of commits in both projects, chances are the architecture might be inaccurate for older commits, as in reality architectures evolve together with the project. We believe that if an architecture is available throughout a project's history and it is updated together with the code, this will improve results.

The results of evaluations using review ratings are not very promising, as their *hit ratio* value is about 50% lower than of those that do not use it. Being familiar with the development process, we see two main reasons for this:

1. Developers use a lot of (their own) recently developed and therefore not yet reviewed methods in their commits.
2. The developers developed the study objects without the help of our system, thus we would hope the results could be better if the developer's could have used our system to find reviewed methods that fulfilled their requirements.

Judging from these results it is questionable if there is any value to using review ratings based on the LEvD process to find better reusable methods in our code search system.

Overall we are content with the obtained results, as they indicate that our code search system works well under an auto-completion scenario, with more than every second recommendation set containing the expected result in our evaluation scenario. If this result can be transferred to real development, then the tool should be useful to a developer and assist him in finding the methods he is looking for.

RQ 2 - Which combination of metrics provides the best results?

Combining the techniques for result retrieval seems to be of limited benefit according to our results. Using all techniques together results in the worst *hit ratio* for *ConQAT* and for the second worst *hit ratio* for *JabRef*, leaving a lot of room for improvement. The lack of good results for all techniques in combination could be a result of weighting the metrics equally, where possibly they should be weighted differently depending on the queried project and general usefulness. More work in this area is required to find out if weighting can make this approach more useful.

5.4 Threats to Validity

The results are subject to a number of threats. First, we use a new evaluation method, which has not been proven to correlate with real world experiences. It would be useful to evaluate our system in a study with real developers who work on a project using our system for a while, to show if our evaluation correlates with reality. Our evaluation method closely follows the actual development of a given study object, thereby mitigating this risk.

As with every automatic evaluation, we have to assume that the developers chose the best possible method available and reused methods where possible. There might have been cases during evaluation where our system actually recommended a better or just as good solution than the one used by the developer, but we cannot measure this without a study questioning the developers.

The results involving the architecture analysis metric are greatly dependent on the quality of the used architecture. Therefore especially the results obtained with *JabRef* might be inaccurate, as the architecture used has been reconstructed by us, without a deep understanding of the code. By using actual dependencies found in the code to create the architecture, we made sure that we did not introduce any nonexistent constraints. A second problem, which affects both used architectures, is that we only have one revision of each architecture. For best results we would need the architecture to evolve together with the code, in the best case developed and updated in the same repository in parallel to the code.

The study objects we evaluated were not built using our system. Therefore we cannot assume that the study object is in the same state as it would have been when building it using our system. If built using our system, we would hope to obtain a better *hit ratio*, as the developers should have used our system at least a few times to find matching functions in cases where they would not have found the best match by hand.

We have only run the evaluation on two study objects: *ConQAT* and *JabRef*. Since these are two Java systems, we cannot yet generalize the results to other systems with much certainty. More study objects need to be evaluated to allow more trust in the results. Finally, both study objects are of medium size with 100,000 - 300,000 lines of code, making the results hard to generalize for smaller and bigger systems or those that are written in a different programming language.

6 Conclusion

This chapter gives a conclusion of the work performed in this thesis.

We have designed a code search system which uses *Context-Dependent Method Recommendation*, *Signature Matching* (section 2.4), *Architecture Conformance* (section 2.9) and *Review Rating Matching* (section 2.3) to find reusable methods for the programmer. We have identified these techniques to be useful for identifying reusable code in response to research question 1 (*What are useful metrics and techniques for a code search system?* – section 1.4) following a comprehensive study of state of the art systems in chapter 3.

The system was designed to be used in multiple usage scenarios like an IDE auto-completion and a web-based search interface, by providing an open interface for clients to query using a special query language (in our case the *Lucene* query language).

Implementation

To answer research question 3 (*Can a shallow parser replace a full parser in code search engines?* – section 1.4) we started of implementing the system based on the shallow parser (section 4.3.3) to see which parts of the system could be implemented. In conclusion the shallow parser can be used to create a basic index of method declarations in the system, but due to its technical limitations it is not useful for extracting extended type information and indexing contexts

Next we implemented the code search system entirely using a full parser (section 4.3.3), providing us with all information necessary. This limits our current implementation to the Java programming language, but allowed us to implement all features we proposed in the system's design (chapter 4).

As a proof-of-concept we implemented a simple *Eclipse* plug-in (section 4.3.6) which adds a new auto-completion provider. This auto-completion provider uses our system to retrieve results and display them in the *Eclipse* auto-completion dialog for the developer to use.

Evaluation

To evaluate our system and to answer research question 2 (*Which combination of metrics provides the best results?* – section 1.4) we proposed a new evaluation design (chapter 5), based on the incremental approach we already used for our main system. The proposed system can be used to evaluate any type of system which provides auto-completion recommendations.

We proposed reading the repository commit by commit and extracting all method calls in the process. For each of these method calls we proposed to send a query to our system containing the calls context, the expected return type at the calls position and the files review rating and package. Our code search system then returns a result set. We then measure the ratio of times we find the method call the developer actually chose to the number of result sets retrieved. This gives us a measure of how often the system would have returned the developers choice and could have thereby helped him during the development process.

We ran our evaluation on the entire development history of *JabRef* and on around 5000 commits of *ConQAT* development history. In conclusion we found that depending on the system under evaluation it is either best to use only *Context-Dependent Method Recommendation* or *Context-Dependent Method Recommendation* plus *Signature Matching* for return types. Using architecture analysis or review ratings in the query provided no improvement in hit ratio.

With hit ratios of 54.71% (table 5.9) for *ConQAT* and 50.21% (table 5.5) for *JabRef*, when retrieving a result set of 5 results, we believe that our code search system provides help for developers in a real world development context. A hit ratio of over 50% means that more than every second set of results retrieved contains at least one result solving the programmers current problem.

7 Future Work

Many components of our code search system could be improved in the future: The analysis gathering the data only works with Java source code, leaving room for expansion to many other languages. Being able to instrument languages like C, C++ or C# could foster adoption rates especially in the industry, as these languages are widely used[1] for commercial and open-source projects.

Expanding the system to work with dynamically typed languages such as *Python*, *Ruby* or *Javascript* would be an interesting task, especially because they are widely used in many open-source projects[2]. As static typing is missing in these languages, signature matching becomes much harder to do, because it is not clear which types are to be used in a given method. Architecture conformance, context matching and review ratings could be used in these languages, so it would provide a great extension to our work.

Besides extending the system to work with multiple programming languages, one could also work on the data back-end. We believe that implementing more code reusability metrics, such as those discussed in section 2.5, could provide a significant benefit when searching for reusable methods, because many methods offering similar functionality are present in large code bases. Ordering them by reusability makes sense and keeps the cognitive distance [24] for reuse low.

Currently we use the software architecture to avoid reuse of methods whose usage would break the software's architecture as defined by the provided architecture specification. One could derive reusability metrics from the architecture specification, for example by ranking methods in components which have a high ingoing/outgoing ratio higher, than those that don't. This stems from the idea, that components which are being accessed from many places in the code base, but don't access a lot of code by themselves, are in a good form to be reused by others.

The system was designed to be able to provide a web based search client. The web based search client would offer an interface allowing the programmer to specify the search parameters by hand and inspect the results on a web interface. We believe this client has a different use-case then a auto-completer: The web based client is more interesting as a tool to find out whether a planned functionality already exists or if it has to be newly implemented. This can be useful in a planning stage, as well as shortly before the programmer starts the actual implementation.

The web client would have to be evaluated in a study, in order to find out if the interface and service actually produce results that are useful to the developer. In this context,

[1] Programming language popularity on *langpop.com*: `http://langpop.com/` (last accessed October 14, 2013)

[2] Github programming language statistics: `http://pxhr.blogspot.de/2013/07/githubs-statistics-programming-languages.html` (last accessed October 14, 2013)

it could be interesting to find out how many "wrong" results (as in searches without a useful hit) can be displayed, before the developer loses interest in the tool and decides to not use it again in subsequent searches.

We have provided a prototypical implementation of an auto-completion plug-in for *Eclipse*. The implementation is very basic, only provides the user interface and does not yet allow inserting code automatically. Implementing this functionality would be an interesting task for the future, as it comes with a host of challenges. One of our goals was allowing the programmer to search code in the repository, that is not checked out locally. This makes auto-completion difficult, as the programmer might not have the code for a given result available locally when using it. This means the plug-in needs to take care not only of inserting the correct method call, but also of setting up the project in the programmer's environment, making sure the code can actually compile after inserting the method call.

With a working auto-completion one could evaluate the system together with users in a study. This would require developers to work with the system for a certain amount of time either on their normal work or on specific projects designed to make evaluation more precise and save time. Both variants could provide interesting insight into whether or not the system is actually useful and delivers as promised.

Bibliography

[1] Awny Alnusair and Tian Zhao. Component search and reuse: An ontology-based approach. In *IRI*, pages 258–261. IEEE Systems, Man, and Cybernetics Society, 2010.

[2] S. Bajracharya, J. Ossher, and C. Lopes. Sourcerer: An internet-scale software repository. In *ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*, pages 1–4. IEEE, May 2009.

[3] Sushil Krishna Bajracharya and Cristina Videira Lopes. Analyzing and mining a code search engine usage log. *Empirical Software Engineering*, 17(4-5):424–466, 2012.

[4] Judith Barnard. A new reusability metric for object-oriented software. *Software Quality Journal*, 7(1):35–50, 1998.

[5] Veronika Bauer, Lars Heinemann, Benjamin Hummel, Elmar Jürgens, and Michael Conradt. A framework for incremental quality analysis of large software systems. In *ICSM*, pages 537–546. IEEE Computer Society, 2012.

[6] Moritz Marc Beller. Static Validation of ConQAT Architecture Descriptions. 2011.

[7] Marcel Bruch, Martin Monperrus, and Mira Mezini. Learning from examples to improve code completion systems. In *ESEC/SIGSOFT FSE*, pages 213–222. ACM, 2009.

[8] Scott C. Deerwester, Susan T. Dumais, Thomas K. Landauer, George W. Furnas, and Richard A. Harshman. Indexing by Latent Semantic Analysis. *JASIS*, 41(6):391–407, 1990.

[9] Florian Deissenboeck, Lars Heinemann, Benjamin Hummel, and Elmar Jürgens. Flexible architecture conformance assessment with ConQAT. In *ICSE (2)*, pages 247–250. ACM, 2010.

[10] Florian Deissenboeck, Elmar Jürgens, U Hermann, and T Seifert. LEvD - A lean evolution and development process. 2007.

[11] Amit Deshpande and Dirk Riehle. The Total Growth of Open Source. In *OSS*, pages 197–209. Springer, 2008.

[12] Michael E. Fagan. Design and Code Inspections to Reduce Errors in Program Development. *IBM Systems Journal*, 15(3):182–211, 1976.

[13] Michael E. Fagan. Advances in Software Inspections. *TSE*, 12(7):744–751, 1986.

[14] Maurice H. Halstead. *Elements of Software Science*. Elsevier, 1977.

[15] Lars Heinemann. *Effective and Efficient Reuse with Software Libraries*. PhD thesis, Technische Universität München, 2012.

[16] Lars Heinemann, Veronika Bauer, Markus Herrmannsdoerfer, and Benjamin Hummel. Identifier-Based Context-Dependent API Method Recommendation. In *CSMR*, pages 31–40. IEEE, 2012.

[17] Lars Heinemann, Florian Deissenboeck, Mario Gleirscher, Benjamin Hummel, and Maximilian Irlbeck. On the Extent and Nature of Software Reuse in Open Source Java Projects. In *ICSR*, pages 207–222. Springer, 2011.

[18] Oliver Hummel and Colin Atkinson. Using the Web as a Reuse Repository. In *ICSR*, pages 298–311. Springer, 2006.

[19] Oliver Hummel, Werner Janjic, and Colin Atkinson. Code Conjurer: Pulling Reusable Software out of Thin Air. *IEEE Software*, 25(5):45–52, 2008.

[20] I. Jacobson, M.L. Griss, and P. Jonsson. *Software Reuse: Architecture, Process and Organization for Business Success*. Addison-Wesley, 1997.

[21] Elmar Jürgens. *Why and how to control cloning in software artifacts*. PhD thesis, Technische Universität München, 2011.

[22] Elmar Jürgens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. Do code clones matter? In *ICSE*, pages 485–495. IEEE, 2009.

[23] Rainer Koschke. Survey of Research on Software Clones. In *Duplication, Redundancy, and Similarity in Software*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), 2006.

[24] C.W. Krueger. Software Reuse. *ACM Computing Surveys*, 24(2):131–183, June 1992.

[25] Otávio Augusto Lazzarini Lemos, Sushil Krishna Bajracharya, Joel Ossher, Paulo Cesar Masiero, and Cristina Videira Lopes. Applying test-driven code search to the reuse of auxiliary functionality. In *SAC*, pages 476–482. ACM, 2009.

[26] Wayne C. Lim. Effects of Reuse on Quality, Productivity, and Economics. *IEEE Software*, 11(5):23–30, 1994.

[27] Thomas J. McCabe. A Complexity Measure. *TSE*, 2(4):308–320, 1976.

[28] Doug McIlroy. Mass-Produced Software Components. In *Proceedings of NATO Software Engineering Conference*, pages 138–155, 1968.

[29] Gail C. Murphy, David Notkin, and Kevin J. Sullivan. Software Reflexion Models: Bridging the Gap between Design and Implementation. *TSE*, pages 364–380, 2001.

[30] Jeffrey S. Poulin. Measuring software reusability. In *Software Reuse: Advances in Software Reusability, 1994. Proceedings., Third International Conference on*, pages 126–138, 1994.

[31] Chanchal Kumar Roy and James R. Cordy. A Survey on Software Clone Detection Research. *Schoo of Computing TR 2007-541, Queen's University*, 2007.

[32] Harvey P. Siy and Lawrence G. Votta. Does the Modern Code Inspection Have Value? In *ICSM*, 2001.

[33] Harry M Sneed, Richard Seidl, and Manfred Baumgartner. *Software in Zahlen die Vermessung von Applikationen*. Hanser, 2010.

[34] Yunwen Ye and Gerhard Fischer. Reuse-Conducive Development Environments. *Automated Software Engineering*, 12(2):199–235, 2005.

[35] Amy Moormann Zaremski and Jeannette M. Wing. Signature Matching: A Tool for Using Software Libraries. *TOSEM*, (2):146–170.