Trust this Code?

Improving Code Search Results through Human Trustability Factors

Bachelorarbeit
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Florian Sebastian Gysin

24. März 2010

Leiter der Arbeit

Prof. Dr. Oscar Nierstrasz

Adrian Kuhn

Institut für Informatik und angewandte Mathematik

Further information about this work and the tools used as well as an online version of this document can be found under the following addresses:

Florian Sebastian Gysin
flo.g@students.unibe.ch
`http://scg.unibe.ch/projects/archive/bender`

Software Composition Group
University of Bern
Institute of Computer Science and Applied Mathematics
Neubrückstrasse 10
CH-3012 Bern
`http://scg.unibe.ch/`

Abstract

The promise of search-driven development is that developers will save time and resources by reusing foreign code in their local projects. To efficiently integrate this code, users must be able to trust it, thus besides relevance of code search results their *trustability* is important as well. In this paper, we introduce a *trustability metric* to help users assess the quality of code search results and therefore ease the risk-cost-benefit analysis they undertake trying to find suitable integration candidates. The proposed trustability metric incorporates both user votes and cross-project activity of developers to calculate a *"karma"* value for each developer. Through the karma value of all its developers a project is ranked on a trustability scale. We present *JBender*, a proof-of-concept code search engine which implements our trustability metric and we discuss preliminary results from an evaluation of the prototype. Furthermore we discuss findings from the creation of a second prototype—*RBender*—that deals with structured search over dynamically typed code.

Acknowledgements

I would like to express my gratitude to everyone who supported me during the time I was working on this thesis. Only due to your support did I manage to successfully complete this work!

First of all I want to thank my supervisor Adrian Kuhn - this work would not have been possible without him. He supported me by providing new ideas, input and motivation and by sacrificing a lot of time for discussing and working with me.

Prof. Oscar Nierstrasz for giving me the opportunity to write this thesis at the Software Composition Group (SCG) and for his inspirational lectures and his support which contributed a lot to my ongoing interest in computer science.

David Erni for critical input and constructive critics, Niko Schwarz and Martti Nirkko for their help with the numerical evaluation of my data.

The whole SCG staff which contributed to this work with critical comments or with interesting discussions and advice.

All the students that accompanied me, not only during the time of this work but during my whole studies, for the great time we had; be it over lunch, while learning or working in the SCG student pool.

My friends for motivating and supporting me and especially my parents for their unconditional support and for always believing in me.

So long, and thanks for all the fish!

Florian, March 2010

Contents

Chapter 1

Introduction

The promise of search-driven development is that developers will save time and resources by searching for software and reusing the search results in their code. A developer in need of certain functionality would—instead of implementing it himself—search for foreign source code and, after finding a suitable candidate, integrate it in his project. This process is not easy: the sources searched are generally huge and not per se trustable (*e.g.*, the web), therefore integration can be difficult. But when the effort for integration can be minimised the developer saves himself the implementation's work.[1]

However, to support search-driven development it is not sufficient to implement a mere full text search over a base of source code. In code search three major concerns have to be addressed, all regarding the results of the code search process:

- Result relevance — does the result fulfil the user's need?
- Result suitability — is the result suitable for integration?
- Result trustability — can the result be trusted?

Previously published work concerning code search engines (CSE) is mainly focused on result relevance, i.e. providing the CSE user with the most relevant results. It is not sufficient though to just provide search results that meet the user's specification and are thus relevant. Human factors have to be taken into account as well. In order to increase the efficiency of search-driven development a search result must also be suitable and trustable [23]. Through this the user saves time trying to assess whether he can trust the foreign source code found by the search engine. If the foreign code is also suitable for the developer's local environment he can proceed to integrate it, thus solving his initial problem.

There are several ways of increasing the trustability of search results. The easiest and most straightforward approach would be to read through the found source code, and to test it properly. After careful examination the user's trust in the

[1]In the text of this thesis we use the masculine form for examples of generic users, developers, etc. This is done solely for reasons of simplicity, no floccinaucinihilipilification is intended.

foreign code should have increased considerably. However, as we are looking for efficient ways of assessing trustability this is obviously not the way to go. What we need is to provide a faster way to let the user estimate the trustability of search results. Another way of increasing the user′s trust in search results is to provide him with suitable metadata. As an example the user would probably trust source code when he knows that the person who wrote that code is a renowned and experienced developer.

The issue of providing meta-information alongside search results and thereby increasing trustabilty has not been widely studied. We are addressing this with our work where we focus on why we think trustability of results is important and how it could be improved. Parts of this work have previously been published in the form of an extended abstract and a short paper [14, 15].

1.1 Approach in a Nutshell

In this thesis we focus on the *trustability* of search results. Relevance of code search results is of course paramount, but trustability in the results can be just as important. Before attempting to integrate a search result into the local code base, the developer first has to assess its trustability in order to take a go-or-no-go decision. A well-designed search interface should allow its users to take this decision on the spot. Gallardo-Valencia et al. found that developers often look into human rather than technical factors to assess the trustability of search results [11]. For example developers will often prefer results from well-known open source projects over results from less popular projects.

In this thesis we present a *trustability metric* for search results. This trustability metric is based on human and social factors. We use data collected from Web 2.0 platforms to assess the trustability of both projects and developers. Our trustability metric is based on collaborative filtering of user votes and on the cross-project activity of developers. For example, if a little-known project is written by developers who also contributed to a popular open source project, the little-known project is considered to be (almost) as trustable as the popular project.

As a feasibility study, we implemented the trustability metric in *JBender*, a proof-of-concept code search engine. The index of our *JBender* installation currently contains trustability assessments for over 3,700 projects, based on 193,000 user votes and the cross-project activity of over 56,000 developers. This thesis also discusses results from an evaluation of the *JBender* prototype are discussed.

A second prototype of a code search engine, *RBender*, focused on code search over dynamic languages. Its goal was a first evaluation of the feasibility of structured code search over dynamic code.

1.2 Contributions of this thesis

Contributions of this thesis are as follows:

- We argue in favour of a stronger focus on the *trustability* of code search results and we explain why we think it is important.
- We introduce a *trustability metric* for software projects. The trustability metric is based on human factors, and uses collaborative filtering of both user votes and cross-project activity of developers.
- We present *JBender*, a proof-of-concept implementation of our trustability metric and discuss preliminary results from an evaluation of the prototype.
- We present findings from the making of *RBender*, a proof-of-concept code search engine able to parse dynamic source code and provide a structured search over it.

1.3 Structure of the Thesis

The remainder of this thesis is structured as follows: Chapter 2 discusses background and related work. Chapter 3 explains the basics of code search engines. Chapter 4 introduces our trustability metric. In Chapter 5 and Chapter 6 we present *JBender* and *RBender*, two proof-of-concept prototypes of code search engines. Chapter 7 evaluates the earlier proposed trustability metric and discusses results from an evaluation of the mentioned prototype. Eventually, we conclude in Chapter 8 with remarks on future work.

Chapter 2

Background and related work

Since the rise of Internet-scale code search engines, searching for reusable source code has quickly become a fundamental activity for developers [4, 32]. However, in order to establish search-driven software reuse as a best practice, the cost and time of deciding whether to integrate a search result must be minimized. The decision whether to reuse a search result or not should be taken quickly without the need for careful (and thus time-consuming and costly) examination of the search results.

Trustability is a big issue for reusing source code. For a result to actually be helpful and serve the purpose originally pursued it is not enough to just match the entered keywords. It is essential that the developer know at least the license under which certain source code was published. Otherwise he will not be able to use it legally. Furthermore, it is very helpful to know from which project a search result is taken when assessing its quality. User studies have shown that developers rely on both technical and human clues to assess the trustability of search results [11]. For example developers will prefer results from well-known open source projects over results rom less popular projects.

2.1 Previous Work

In recent years special search engines for source code have appeared, namely GOOGLE CODE SEARCH[1], KRUGLE[2] and KODERS[3]. These code search engines (CSE) all focus on full-text search over a huge code base, but lack detailed information about the project. Search results typically provide a path to the version control repository but little meta-information on the actual open source project; often, even such basic information as the name and homepage of the project are missing.

[1]`http://www.google.com/codesearch`
[2]`http://www.krugle.org`
[3]`http://www.koders.com`

SOURCERER[4] by Bajracharya et al. [5] and MEROBASE[5] by Hummel et al. [18] are research projects with internet-scale code search-engines. Both provide the user with ways of structured code search. In terms of trustability both of these code search engines provide the developer with license information and project name of the found search results. Merobase also provides a set of metrics such as cyclomatic and Halstead complexity. An improved version of Sourcerer with trustability data is in development, though it has not yet been published[6].

In addition to the web user interface, both Sourcerer and Merobase are also accessible through Eclipse plug-ins that allow the developer to write unit tests. These are then used as a special form of query to search for matching classes/methods, *i.e.*, classes/methods that pass the unit tests [18].

CODE GENIE is the Eclipse plug-in by Lemos et al. based on the Sourcerer code search engine [25]. It focuses on directly using test cases written during test-driven development to search for possible result candidates.

CODE CONJURER is the Eclipse plug-in based on Merobase, focusing on lowering the barrier for software reuse in search/reuse-driven development [18, 21]. Using unit tests as form of query is a way of increasing technical trustability: unit-tested search results are of course more trustable, although at the cost of a more time consuming query formulation (*i.e.*, additionally writing the unit tests).[7] The kind of results returned are also limited to clearly-defined and testable features. A combination of technical trustability factors (*e.g.*, unit tests) and human trustability factors might be promising future work.

Reiss et al. developed S-6[8], a prototype of a meta code search engine [28, 29]. It is based on top of Google Code Search. The focus lies on exact and meaningful specification of structured search over source code. S-6 also provides unit tests as search criteria.

Inoue et al. presented a ranking model that could be used to rank software components by significance [20]. Significance in their ranking is based on use relations of components. They also presented a search system named SPARSE-J based on this ranking.

There have also been efforts to set conventional standards for code search engines and software repositories. Garcia et al. compiled a survey of previous efforts in research and industry CSEs and identified requirements that state-of-the-art search engines should meet [12].

[4]`http://sourcerer.ics.uci.edu`
[5]`http://www.merobase.org`
[6]Personal communication with Sushil Bajracharya.
[7]For more information on the subject of unit testing as a form of query formulation see 3.2.2.
[8]`http://www.cs.brown.edu/~spr/research/s6.html`

2.2 Related Work

We are not the first to use collaborative filtering in code search. Ichii et al. used collaborative filtering to recommend relevant components to users [19]. Their system uses browsing history to recommend components to the user. The aim was to help users make cost-benefit decisions about whether or not those components are worth integrating. Another approach is the work of Ohira et al. [26]. To enhance cross-project knowledge collaboration they used collaborative filtering to match cross-project knowledge and skills of developers in free/open source project development.

Among other things our approach is based on commit data from project repositories. Arafat et al. have studied the distributions of commit sizes and commit categories to open source projects [3] and the comment density therein [2].

Patterson wrote about the experience of writing a search engine and about the pitfalls that await developers who attempt the same [27]. It constitutes a valuable first overview of the subject of writing your own search engine.

Starke et al. have worked on relevance of code search results [33]. They especially investigated how effectively users can specify the information they are seeking and how they work with the thereafter delivered search results. One of their conclusions argues that ranking code search results correctly—possibly by "confidence values"—could be of great value to developers.

Technical trustability factors have been covered by Lemos et al. [24] in the form of unit tests. If a search result passes the test which was formulated as part of a query, this can help to assert runnable and working search results.

Dynamic programming languages have a rather small role in the software development world today. But importance will increase over the next years and may well be critial to next-generation application development [9]. We created the *RBender* prototype with the aim to provide structured search over dynamic code in anticipation of this trend.

Corbat et al. have worked on a Ruby refactoring extension for the RDT[9] plug-in for the Eclipse IDE [8]. Their work also includes parsing Ruby source code and interpreting the thereby obtained AST (abstract syntax tree). We used part of this parsing functionality for the *RBender* project.

[9]Refactoring for Ruby Development Tool - `http://r2.ifsoftware.ch/trac`

Chapter 3

Code Search Engines

In this chapter the general idea of a code search engine (CSE) is presented and different features of CSEs are explained. Also we discuss how CSEs can be improved to be more efficiently operable by users.

3.1 The Code Search Engine in a Nutshell

The basic idea of a code search engine (CSE) is to let the user search for source code and to present him results for his query. Similar to a classic text search engine a CSE gathers a base of source documents and creates an efficiently searchable index over it. In the case of the CSE these documents, *i.e.*, the body over which to search, are source code files. Most CSEs use a full text search approach where a certain query string is parsed and searched in the source code documents that were indexed for the CSE; in this way most CSEs work the same way as conventional full text search engines.[1]

3.2 Improving Search Efficiency

When investigating the efficiency of a CSE and the search tasks that can be accomplished with it several ways to improve the efficiency come in mind:

Code base A code search engine's value increases as its underlying code base increases, providing a greater range of hopefully better matching results even for complicated and narrow searches.
$\Rightarrow$ Provide *more* results.

Result relevance The search strategy, which determines how the possible results are searched for, can be improved. Through this result relevance is

[1]Of course the parsing of the query string and the source documents differs greatly: source code and natural language call for different stop-word lists, word-stemming, *etc.*. A good source of information about search engines and algorithms can be found in Grossman and Frieder [13].

increased. On the one hand this includes the algorithms of the actual search engine, on the other hand the possibilities the user has to interact with the search engine, *i.e.*, to specify his query.
⇒ Provide *relevant* results.

Result suitability An increase in the suitability of results makes it easier for the developer to integrate them. A CSE could adapt found results to the local environment, *e.g.*, adapt the code formatting style to local conventions.
⇒ Provide results *suitable* for integration.

Result trustability Increasing the trustability and therefore the quality of the search results provided for a search lowers the time it takes the user to make decisions and to actually implement the found source code.
⇒ Provide *trustable* results.

3.2.1 Increasing the number of potential results

The power of a search engine relies heavily on the magnitude of the underlying index. A code search engine needs a huge code base to index. Only through that can it provide a big enough solution space for a sufficient level of possible code searches. For internet scale CSEs this source code is generally collected by crawlers from big open source repositories. Examples for such repositories are Sourceforge[2], Github[3], RubyFore[4] or Tigris[5]. The main selling point of these repositories is—apart from the easy access—that the contained source projects are distributed under open source licenses. This makes them actually suitable as results for search and reuse based development. See Subsection 3.2.3 for more about licenses of code search results.

3.2.2 Increasing result relevance

In order to examine the efficiency of a CSE we must start at the beginning of a search task: let's think about the actual need a user wants to satisfy when using a code search engine. When a user performs a source code search he does search for source code, but what he is actually looking for is functionality of some sort or another: for example a framework, library, class or method performing a certain task [11].[6]

This is in contrast to "normal" full text search (*i.e.*, via Google or Yahoo) where the user is generally looking for a piece of prose. For source code search this leaves a gap between what the user is searching and what he is looking for. To enable efficient code search this gap must be overcome by letting the user

[2]http://sourceforge.net
[3]http://github.com
[4]http://rubyforce.org
[5]http://www.tigris.org
[6]That is when the user is a developer and not a researcher performing an empirical study *on* source code.

specify the functionality he is looking for as precisely as possible. This is a task that is hardly accomplished by search engines that use pure full text search approaches: as source code is a highly structured form of text we should use this underlying structure and with it the implicit information to the search engine's advantage.

Structured code search

One of the basic approaches to increase result relevance is structured code search [31]. Structured code search does not treat the underlying source code as simple text but respects its characteristics. This means structured code search takes advantage of the structured form of source code and works with the thereby specified functionality. This way the user can more precisely specify the functionality he is looking for, thereby increasing result relevance. Also the result for a structured code search is not limited to an indexed source file as with conventional full text approaches, but can be any possible logic entity: a library, a class, a method, ...

An example of structured code search is the specification of a method signature which our results should satisfy:

A user is searching for a function which converts integer values into roman numerals.[7] *Instead of just searching a code base with the string "roman numeral", a structured code search approach would let the user—additionally—specify that he is looking for a method that takes an integer as the single parameter and returns a string.*

The CSEs providing structured code search approaches today are mainly limited to static programming languages. It is clear that explicitly and statically typed languages supply much more information that is easily exploitable by a search engine when compared to dynamically typed languages. In our opinion it is however feasible as well to invest in structured code search for dynamically typed languages. They also greatly profit from a more precise query specification and with the tools in development the analysis of dynamic languages is becoming more and more promising [8, 30, 17]. We focused on the aspect of structured search over dynamically typed languages with the work on our prototype *RBender*: it is a CSE which provides structured code search over Ruby, a dynamically typed language. See Chapter 6 for more information concerning *RBender*.

Unit testing as query formulation

Unit tests provide a way of asserting certain functionality in source code [7]. It follows that unit tests contain a lot of behavioural and structural information about the source code they are testing. Specifying unit tests as a way of formulating queries can thus be seen as a special form of structured code search (3.2.2). This means that a user of a CSE will write one or more unit tests as a part of his query, possibly alongside a conventional full text search. The code search

[7]This function would for example return VII for 7 or XXIV for 24.

engine then runs these unit tests on all possible result candidates (generally the results from the full text search); only the results that "pass" the unit test(s) are presented to the user.

Let us reconsider the example from above (see 3.2.2):

Our user is still searching for a function which converts integer values into roman numerals. As he is using a CSE that offers unit tests as a form of query formulation he specifies his query as follows: he uses the string "roman numeral" for a full text search and additionally writes a unit test. The method he is looking for must return the string "VII" when fed the integer '7', string "XXIV" for integer '24' and string "CLVI" for integer '156'. The CSE now provides him only with the results that not only match his query string, but also comply with the specified unit tests.

Unit tests are an effective way of ensuring result relevance for code search [28, 29, 25]. What also makes them attractive is the fact that unit tests are—or should be—written before code implementation [7]. This means a developer who has set up his unit tests in a test-driven development approach could then just start to fill the "blanks" with search results provided to him by a CSE. The main disadvantage of unit tests as search queries is that test execution is rather time consuming. The work of Hummel et al. has shown that code search with unit tests can possibly take a very long time, thus making it practical only in certain situations [18]. Given this limitation it is only feasible to use unit test queries as a final means of filtering the results provided by less time-consuming forms of code search (conventional full text search, structured code search).

3.2.3 Increasing result suitability

Suitability of code search results is a measure for how suitable these code search results are for actual implementation. For a result to actually be helpful and serve the purpose originally pursued with the search it is not enough to just match the entered keywords. It is essential that the developer know at least the license under which certain source code was published; otherwise the developer will not be able to use the code legally. Furthermore the impact of the piece of source code on its environment should be as small as possible: no developer wants to integrate a piece of source code into his project when this source code is relying on half a dozen additional third party libraries.

But there are some more ordinary factors that play into suitability as well: basically every factor that is an obstacle the developer has to overcome in order to integrate potential integration candidates decreases suitability. This ranges from a simple difference in code formatting and naming conventions to discrepancy in functionality, *e.g.*, a different error handling. Reiss suggested that to maximize suitability of code search results a code search engine can provide ways of adapting search results to the local source code—according to the specification of the developer [28, 29]. These adaptations—as the problems they try to tackle—range from simple changes like reformatting or renaming to more complex operations like API transformation and adjustment to existing dependencies. Code search engines that provide good examples for improved result suitability are Code Conjurer and S-6 (see "Previous Work" in Section 2.1).

3.2.4 Increasing result trustability

The trustability of code search results plays a major role in efficient search-driven development. When a user integrates foreign code into his local project he basically has only two options: he can either trust the foreign code and integrate it as is; or he can try to work through it, comprehending its functionality and finally figuring out if it fits the purpose intended. The latter of the two options is obviously very time consuming: it implies reading and reverse-engineering foreign code. The effort of manually assessing the foreign code may well be so high that it is not feasible for the user to integrate it at all but must implement the needed functionality himself. It is the aim of search driven development to save the developer time and resources, hence one must prevent similar scenarios. It therefore follows that the user is in need of another way of assessing trustability of his code search results.

What we propose is to display additional trustability information to the user alongside the code search results. This abstract trustability information—displayed in a reasonable format—will help the user efficiently assess the trustability of his search results while browsing them. Thus the additional information the user is confronted with should have an easily comprehensible form and should in no way overburden the user with data.

Trustability of source code can be split into two major categories: technical and human trustability factors. The following two sections clarify this distinction.

Technical trustability factors

Technical trustability factors are intrinsic properties of the source code that make it more trustable for a potential user. In other words technical trustability factors are software metrics that suggest increased quality of the software in some way. This could be for example the test-coverage of a library/project: a high test-coverage compared to a low test-coverage indicates a certain level of diligence observed by the developers while working on this project. Furthermore successful unit testing asserts certain fundamental behaviour of the tested code and through that make it more trustable. These technical trustability factors are internal properties of the actual source code; they can be collected and—after refinement—be presented to a code search user.

A few examples of technical trustability factors are

- test coverage of libraries;
- test coverage of a specific method/class;
- comment-line source-line ratio in the source code; or
- code cohesion.

Human/social trustability factors

By far not all information that makes source code trustable for a developer is contained in the source code itself. Recent research showed that source code alone is often not sufficient to assess the trustability of a search result: one must take into account human and social factors [11, 23]. We must not forget that the quality and consequently the trustability of a piece of source code mainly depends on who it was written by: For example, we would probably trust a piece of code written by a close colleague because we have some knowledge of his work. Or we would trust a piece of code that was under its team's development for a long time, since the many revisions would probably have solved all/most of the errors. What makes foreign source code untrustable per se is that developers lack the meta information implied by the two above examples. To gain trust in foreign code a developer must somehow be provided with this metadata.

Human and social trustability factors are derived from meta information to a certain piece of source code. For example:

- Who wrote a certain piece of code/certain project?
- Does this person write good/trustworthy code?
- How do others think about this person?
- How many people use this specific code/project?
- How long has this code/project been in development?
- How regularly is this code/project updated/worked on?
- Is the code/project running actively or does it come from a dead branch?

The problem of human/social trustability factors lies mainly in the fact that they are not derived from the actual source code but rely on additional external data. This means that it is no longer sufficient for a CSE to crawl the web (*i.e.*, large free open source repositories) for source code: often these repositories provide little or no additional data related to the stored source projects. When a CSE wants to provide the user with human trustability information this information must be gathered in some other way. Because the needed meta data cannot be extracted from the source code, this meta data often ultimately relies on user input.

At this time not many projects come with sufficient additional information. This is a limitation that greatly reduces the number of open source projects which could be indexed in a CSE, limiting its source code base drastically compared to CSEs that eschew human trustability information.

What we propose in this thesis is a trustability metric that takes into account several human/social trustability factors and calculates a trust value for each search result. This trust value can then be fed to the user who can for example sort the search results by trustability. Read more about this approach in Chapter 4 where we present our trustability metric, or in Chapter 5 where we present a proof-of-concept prototype implementing said metric.

Presentation of trustability information

After collecting trustability information about the code search results the question arises as to how this information should be presented to the user. Up to now this abstract term of trustability information can consist of percentage values (*e.g.*, test coverage), relations (who developed what?), time spans (time of development), or any other form of metric about indexed source code. Of course it would be of no help to the user when the search interface (or rather the results interface) is overloaded with all this information. Research has shown that even small variations from familiar interfaces or small distractions in the search process can disrupt and confuse a user [16]. Providing the user with an abundance of trustability information for each and every search result is of course out of the question. This calls for a presentation of the trustability related information that is non-intrusive and does not overburden the user interface.

Chapter 4

Trustability Metric

In this chapter, we propose an abstract trustability metric for code search results that uses collaborative filtering of both user votes and cross-project activity of developers. It is our aim that this trustability metric help developers quickly assess code quality while they are browsing through results. In consequence this either raises or refutes the user's trust in search results. To finally assess the trustability of code search results we combine traditional full text search with meta-information from Web 2.0 platforms.[1] In the second part of this chapter a toy example is presented to demonstrate the calculation of trust values.

In a nutshell We consider a project to be trustable if there are significant contributions by developers who have also significantly contributed to projects that have received a high number of user votes.

4.1 Developer Karma

Our trustability metric is based on the notion of developer "karma". We use karma as a measure of trustability/quality of developers, so in effect trustability of a project increases or decreases according to the karma of the developers who worked on it. Our calculation of karma is based on the assumption that developers who have (significantly) contributed to a lot of successful projects have a lot of experience and thus deliver high quality source code. Hence we assign these developers with a high karma value.

The following information is required by our trustability metric in order to calculate developer karma (and ultimately project trustability):

- A matrix $M = (c_{d,p})$ with the number of contributions per contributor d to a project p.

[1]See Chapter 5 for more information about the implementation of the proposed metric in a proof-of-concept prototype CSE.

This records how influential a developer is for a specific project and—summed up—how active a developer is in general.

- A vector $V = (v_p)$ with user votes for software projects.
  This signals the users' trust in projects. Gallardo-Valencia et al. refer to user votes as the number of "fellow users" [11].

We use collaborative filtering of both user votes and cross-project activity of developers. For example, if a little-known project is written by developers who have also contributed to a popular open source project, the little-known project is considered to be as trustable as the popular project. Since both the number of contributions per contributor and the number of votes per project follow a power-law distribution,[2] we use $\log$ weighting and *tf-idf*[3] weighting where applicable.

We define the *karma* K_d of a contributor as

$$K_d = w_d \sum_P c_{d,p} \ln(1 + v_p) \quad \text{where} \quad w_d = \frac{\ln(1 + \mathrm{pf}(d))}{\mathrm{pf}(d)}$$

which is the sum of the votes of all projects (v_p), weighted by the number of contributions to these projects ($c_{d,p}$). We take the $\log$ of the user votes v_p to limit the dominating effect of few projects with a very high number of user votes. The sum is weighted by w_d of the contributor, which is the $\log$ of the project frequency divided by the project frequency.[4] The project frequency is the count of the projects a certain developer contributed to. See Section 4.3 for an exemplary calculation of the karma of a developer.

4.2 Project Trustability

We use the previously defined karma of the involved developers to assess a total trustability of a certain project. Our trustability metric relies on the assumption that contributing developers with high karma endorse trustable high quality projects. Hence if a project has a lot of commits by developers who have a high karma value we consider the project to have a high trustability. Based on this, *trustability* T_p of a project is defined as

$$T_p = \sum_D w_{d,p} K_d \quad \text{where} \quad w_{d,p} = \frac{\ln(1 + c_{d,p})}{\sum_{d' \in D} \ln(1 + c_{d',p})}$$

[2]See Subsection 7.1.2 for more information on the power-law distribution of the user generated data from Ohloh.

[3]"Term frequency-inverse document frequency" is a statistical measure used to evaluate how important a 'word' is to a 'document' in a corpus. The importance increases proportionally with the words appearances in the document but is counterbalanced by the frequency of the word in the corpus. [13]

[4]In earlier versions of this work we used the inverse $\log$ of the project frequency as a weighting. We observed that this still leads to a strong scattering of single results and therefore applied the new weighting.

which represents the sum of the karma of all the projects contributors(K_d), weighted by the number of their contributions. Note that we divide project trustability by the total number of contributions, but not contributor karma. This is done on purpose, as contributors are more trustable the more they commit (based on the assumption that all accepted commits require approval of a trusted core developer, as is common in many open source projects) but projects are not per se more trustable the larger they are.

To summarize, we consider a project to be trustable if there are significant contributions by contributors who have also significantly contributed to projects (including the project in question) that have received a high number of user votes.

The proposed definition of trustability is dominated by cross-project contributors, *i.e.*, contributors who contributed many times to many projects with many votes. This is in accordance with empirical findings on open source that have shown how cross-project developers are a good indicator of project success [22]. This behaviour is also known as "the rich get richer" in the theory of scale-free networks and is considered an inherent and thus common property of most social networks [6]. For an overview of the current ranges of karma and trustability values please see Chapter 7.

4.3 A Toy Example

This section presents a toy example for the trustability metric over a base of 3 projects and 4 developers. We deliberately chose a few numbers to illustrate some of the properties of the above presented metric. Table 4.1 and Table 4.2 show example base data for the calculation of a trustability metric.

Developer	*Foo*	*Bar*	*Qux*
Alice	500	-	300
Bob	100	-	-
Charlie	-	50	300
Dave	50	250	-

Table 4.1: Example data: $(c_{d,p})$ - number of contributions per developer D_i to a project P_j.

Project	V_P
Foo	10000
Bar	400
Qux	50

Table 4.2: Example data: (v_p) - number of user votes per project P_j.

4.3.1 Sample calculation

As a demonstration let us calculate the karma value of *Alice*.

$$K_{Alice} = \frac{\ln(1 + \mathrm{pf}(Alice))}{\mathrm{pf}(Alice)} \sum_P c_{Alice,p} \ln(1 + v_p)$$

A word on logarithms: In the formulas presented we used logarithmic functions. What we are using is the natural logarithm $\ln$ to the base e. Logarithms yield values of zero for $x = 1$ and are not defined for $x = 0$. To avoid problems that could be raised by this—*e.g.*, division by zero—we do the following: we add one within each calculation of a $\ln(x)$, *i.e.*, we calculate $\ln(x + 1)$. This avoids zero values for projects with only one vote, for developers who only worked on one project or for developers who only have a single commit in a project.

As we can gather from the above table *Alice* has worked on 2 projects (project *Foo* and project *Qux*), therefore her project frequency is 2.

$$\mathrm{pf}(\mathit{Alice}) = 2$$

With the values from Table 4.1 and Table 4.2 we calculate the following karma value for *Alice*. (Mind the above mentioned $+1$ operation for $\log$ parameters.)

$$K_{\mathit{Alice}} = \frac{\ln(2+1)}{2}\Big(\ln(10000+1)\ln(500+1) + \ln(50+1)\ln(300+1)\Big)$$

$$K_{\mathit{Alice}} = 43.8$$

Karma for the other developers is calculated similarly.

$$K_{\mathit{Bob}} = 29.5 \quad K_{\mathit{Charlie}} = 25.3 \quad K_{\mathit{Dave}} = 38.1$$

After assigning karma values to all developers we can calculate the trustability of projects according to the metric presented in the last section. Here as an example we calculate the trustability of the project *Qux*.

$$T_{\mathit{Qux}} = \sum_{D} \frac{\ln(1 + c_{d,\mathit{Qux}})}{\sum_{d' \in D} \ln(1 + c_{d',\mathit{Qux}})} K_d$$

Using the example of the project *Qux* we enter the correct values and take the sum over the developers *Alice* and *Charlie*:

$$T_{\mathit{Qux}} = \frac{\ln(300+1)}{\ln(300+1) + \log(300+1)}43.8 + \frac{\ln(300+1)}{\ln(300+1) + \ln(300+1)}25.3$$

$$T_{\mathit{Qux}} = 34.6$$

After processing all example data through the two functions for K_d and T_p we get the trustability data. This gained trustability data consists of the karma values for each developer and the trust value for each project; it is listed in Table 4.3 and Table 4.4.

Developer	K_D
Alice	43.8
Bob	29.5
Charlie	25.3
Dave	38.1

Table 4.3: Example trustability data: The karma values for all developers D_i.

Project	T_P
Foo	37.8
Bar	32.8
Qux	34.6

Table 4.4: Example trustability data: the trust values for all projects P_j.

4.3.2 A few things to note

Let's have a look at the data from the toy example and the results of the conducted trustability calculation:

- *Alice* has the highest karma value. This comes mainly from the fact that she contributed the biggest part of project *Foo* which is a very popular project: it has 10,000 user votes.
- Although *Charlie* and *Dave* have about the same number of commits (350 vs. 400) *Dave* has a much higher karma value. This results from the fact that *Dave* made the biggest part of his contributions to a moderately successful project (project *Bar*; 400 votes), where as *Charlie* contributed mainly to project *Qux* which is less successful (50 votes).
- Project *Qux* has a higher trust value than project *Bar* although it is not as successful. (It has only 50 user votes compared to the 400 votes of project *Qux*.) This is because a significant part of project *Qux*—300 out of the total of 600 commits—was developed by *Alice* who has a very high karma value.
- All projects *Foo*, *Bar* and *Qux* have trust values that are relatively close to each other. This is because in the setup of this example there exists a (comparatively) strong cross-project activity of developers.

4.4 Implementation and Evaluation

We created a proof-of-concept prototype called *JBender* which implements the trustability metric T_p introduced in this chapter. Further details are in Chapter 5. Chapter 7 covers results from an evaluation of *JBender* and the trustability metric implemented therein.

Chapter 5

JBender: A trustability enhanced CSE

In this chapter we present our proof-of-concept prototype *JBender*. *JBender* is a code search engine that provides code search over Java source code. It enhances trustability of the search results delivered by additionally processing meta information and providing the user with a trustability estimate for each search result.

5.1 Motivation

It is our aim that the user of a code search engine is provided with as much information as possible with as little effort as necessary. Therefore *JBender* provides the user with additional meta information along with the actual pieces of code that were found for a query.

We have developed *JBender* as a code search engine which enriches code search results with trustability information. To add to the information content of search results we combine two main sources to form the *JBender* code search engine. On the one hand there is the actual code base of the search engine over which an index is created. On the other hand we have created a database of metadata for the projects in the code base.

5.2 Jbender's Architecture

This section will shortly describe the architecture of *JBender*. *JBender* creates a searchable index over the code base and provides a code search over it. Its novelty however lies in the underlying metadata which is linked to the projects in the searchable code base - upon finding results from the latter *JBender* can supply the meta information stored for the result's originating project.

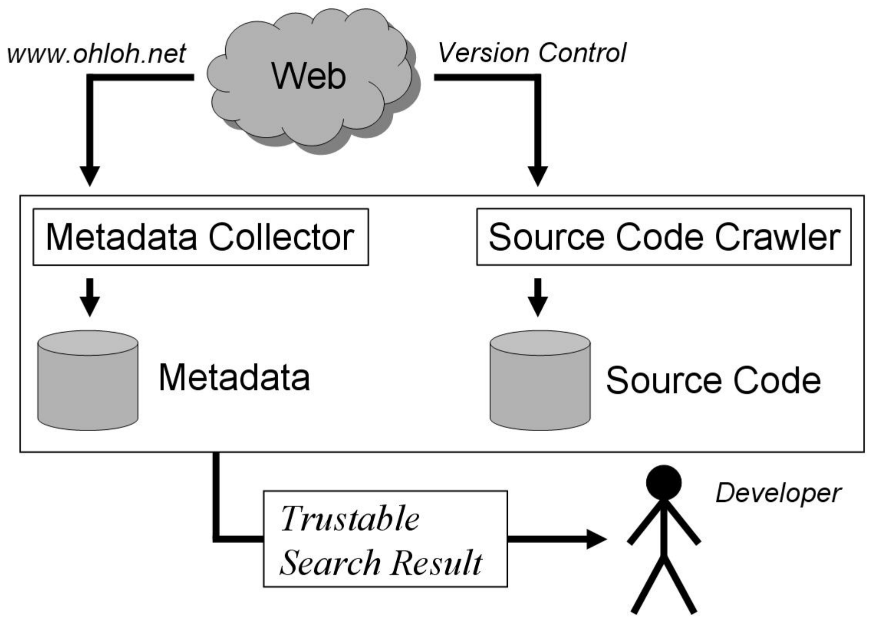


Figure 5.1: Architecture of the *JBender* prototype. *JBender* enhances search results from source code with a trustability estimate that is based on social data collected from the Web 2.0 website *Ohloh*.

This meta information has its origin not in the source code over which the index of *JBender* was created, but is collected from external sources and saved in a metadata base alongside the source code index. See also Figure 5.1 for an illustration.

5.2.1 The Bender core

The core of the *JBender* prototype—*Bender*—is a search engine written in Java. It is an implementation of functionalities provided by the LUCENE[1] framework and provides the most basic functionalities a code search engine must provide. These are:

- Parsing source code files to extract information.
- Analysing this information to create abstract 'documents' to index.
- Creating an index with a term-document matrix.
- Searching an existing index and returning a selection of result documents.

The decision to use the Lucene framework was made on the basis of two main reasons:

[1]Apache Lucene is a text search engine library written in Java. For further information refer to `http://lucene.apache.org/java/docs`.

Parsing Java Source Code It was our goal to create a CSE that was able to parse and index Java source files. As the Lucene Framework is written entirely in Java the core of our search engine could also be written in Java. Using Java as the main programming language made it possible for us to use the Java compiler available from Sun and Eclipse in order to parse Java source files to obtain an AST.

Focus Result Enhancement We have put the focus of this thesis not on the actual creation of the index' search matrix and the algorithms to search therein, but on the enhancement of search results. The Lucene Framework is a tool that provided us with the needed functionality and let us concentrate on critical topics of our search engine.

5.2.2 JBender's metadatabase

Our source of meta data is the OHLOH[2] project. Ohloh is a social networking platform for open source software projects where projects (or rather their developers) can specify additional information. However, Ohloh does not allow users to actually search through or interact with the source code; Ohloh is not a code search engine. Ohloh provides user-contributed information on both open source projects and their developers, composing valuable information for search users. Users can vote for both projects and developers whether and how much they like them by rating projects and giving "kudos" to certain developers. Furthermore kudos are (automatically) given to developers who have worked for successful projects, i.e. projects with large user bases. See Section 7.2 for a more detailed discussion of Ohloh's rankings.

For the *JBender* prototype we collected the trustability meta-information from Ohloh, which provides user-contributed information on both open source projects and their developers.

Metadata stored in the database includes (among others):[3]

- Description of original project
- Project homepage
- Rating of the project
- List of current repositories (type, url, ...)
- Licenses of files in the project
- Employed programming languages (lines of code, comment ratio, ...)
- The project's users and developers who worked on the project (kudos, commits per project, ...).

[2]Find the Ohloh project at `http://www.ohloh.net`.
[3]You can find a full list of the stored metadata in Appendix B.

The above listed data is collected on a per-project basis. This allows for better information density when seen from a trustability point of view: comparing users or ratings would hardly be possible on a per-file basis. There are some problems however that arise through generalisation of project information to single files that are discussed in Subsection 7.3.2.

The amount of data collected per project can be quite extensive. In order not to clutter the user interface by providing too much information we limited the actually visible part of the metadata to a small but important subset. Currently the *JBender* interface displays

- the trust value calculated by our metric;
- the number of users the project has; and
- the numer of developers who worked on the project.

It would be conceivable that in a future version of *JBender* the user could customize the amount of data presented to him and/or the way it is arranged in the GUI. This would give the user more freedom and he could adapt the search engine to his personal needs.

5.2.3 JBenders's codebase

In addition to the collected metadata, *JBender* also follows the links to the version control repositories that are listed on Ohloh. For all accessible repositories *JBender* creates full copies in the form of local Git repositories. The source code files in these local repositories are then parsed and a searchable index is created over them.[4]

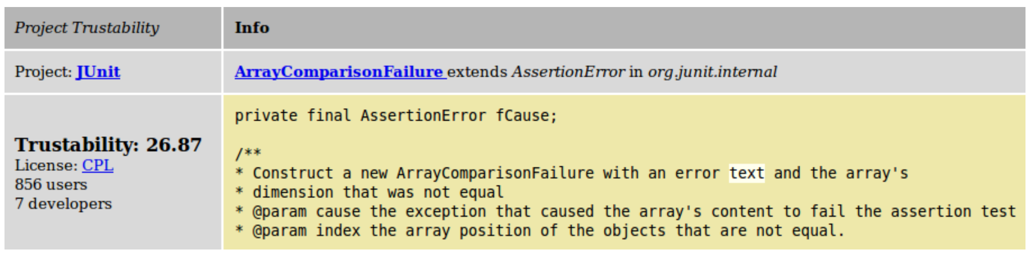
JBender then provides a basic structured code search over various parts of the indexed source code. Examples are method and class names or their bodies, comments, component visibility, dependencies and implemented interfaces. For a full list of the indexed fields in *JBender*'s index see Appendix A.

At the time of writing the code base of *JBender* contains more than 200,000 source code files and a total number of over 3.5 million indexed terms. Through further crawling of the projects listed on Ohloh the number of source files could be increased at least ten-fold.

5.3 Trustability enhanced Results

The results given by *JBender* present additional meta information alongside the source code. Furthermore each result is measured by our trustability metric and

[4]*JBender* only downloads and parses Java source code. We decided to use Java projects because they are numerous and the source code—Java being a statically typed language—is easy to parse. Naturally it would be desirable for a code search engine to 'speak' any possible programing language. However this would not have been feasible for our prototype and would not have promised much additional value on the subject of code trustability.



Project Trustability	**Info**
Project: **JUnit**	**ArrayComparisonFailure** extends *AssertionError* in *org.junit.internal*
Trustability: 26.87 License: CPL 856 users 7 developers	see code below

```
private final AssertionError fCause;

/**
* Construct a new ArrayComparisonFailure with an error text and the array's
* dimension that was not equal
* @param cause the exception that caused the array's content to fail the assertion test
* @param index the array position of the objects that are not equal.
```

Figure 5.2: Screenshot of a *JBender* search result with trustability estimate. On the right there is the actual search result, with full name and code snippet. On the left there is information about the originating project and the trust value calculated by the trustability metric.

assigned a trustabilty value which is also displayed to the user.

The following data from Ohloh was directly used for the trustability metric: as contributors we used the developers of the projects and as the number of contributions we used the number of commits. As user votes we used the number of developers who "stacked" a project, which is Ohloh's terminology for claiming to be an active user of a project. That is, we interpret "votes" as a user expressing his trust in a project by 'stacking' it.

In our case, both users and contributors are open source developers. To be a user, the developers must be registered on Ohloh and create an account. This is not necessary for being a contributor, since that information is taken from version control systems. If a user whose involvement as a developer was already monitored via version control (*e.g.*, as a commiter in SVN) registers on Ohloh, he can 'claim' these existing commits as his own. The two accounts (the newly registered one and the existing "contributor" account) are then merged.

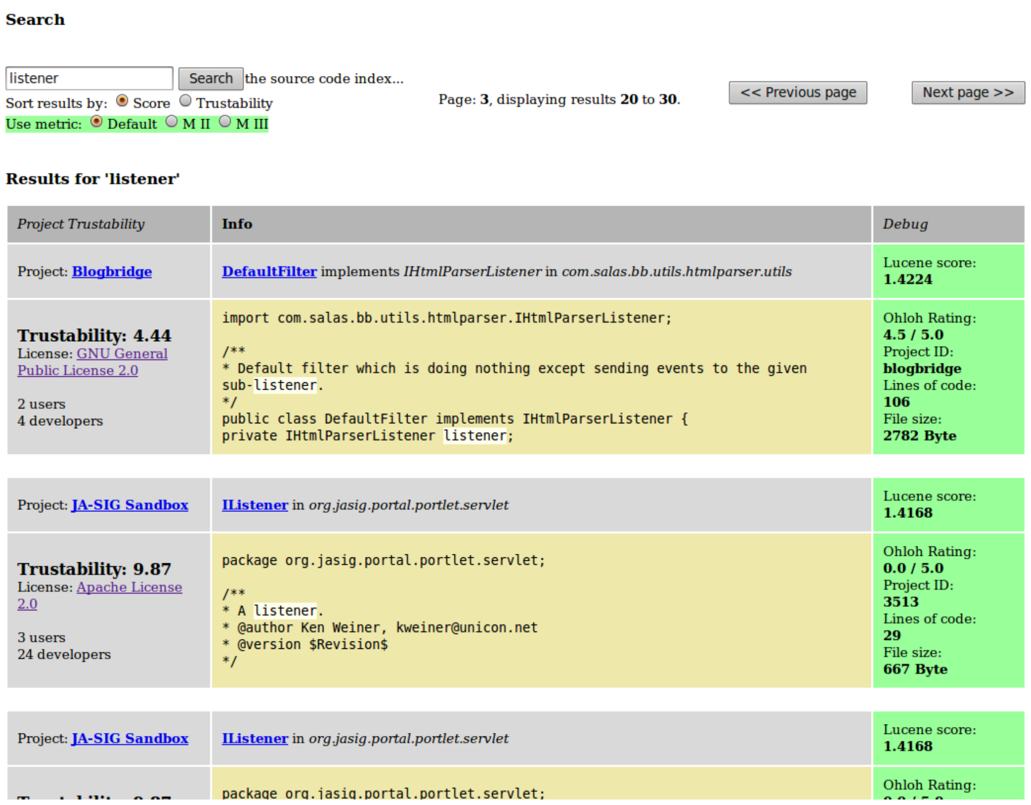
As explained in Chapter 4 this trustability metric takes into account several of the collected meta parameters and calculates a trust metric for each result according to which the results can be sorted.

5.4 JBenders GUI

The layout of *JBender*'s search results is deliberately kept very simple and lucid in order to be efficiently usable. It has been shown that efficient search requires compact and well-arranged interfaces: interfaces which do not burden the user with too much information or a complex information seeking process [16].

For future versions of *JBender* it would be desirable to enable the user to change the default presentation of results, *i.e.*, which metadata is shown and where it is located in the GUI.

Figure 5.2 shows a screenshot of a single search result from *JBender* v1.0.0 as we would present it to a user. On the right there is the actual search result, with full



Search

listener | Search | the source code index...

Sort results by: [✓ Score] Trustability

Use metric: [✓ Default] M II M III

Page: **3**, displaying results **20** to **30**.

<< Previous page | Next page >>

Results for 'listener'

Project Trustability	**Info**	*Debug*
Project: **Blogbridge**	**DefaultFilter** implements *IHtmlParserListener* in *com.salas.bb.utils.htmlparser.utils*	Lucene score: **1.4224**
Trustability: 4.44 License: GNU General Public License 2.0 2 users 4 developers	`import com.salas.bb.utils.htmlparser.IHtmlParserListener;` `/**` `* Default filter which is doing nothing except sending events to the given sub-listener.` `*/` `public class DefaultFilter implements IHtmlParserListener {` `private IHtmlParserListener listener;`	Ohloh Rating: **4.5 / 5.0** Project ID: **blogbridge** Lines of code: **106** File size: **2782 Byte**
Project: **JA-SIG Sandbox**	**IListener** in *org.jasig.portal.portlet.servlet*	Lucene score: **1.4168**
Trustability: 9.87 License: Apache License 2.0 3 users 24 developers	`package org.jasig.portal.portlet.servlet;` `/**` `* A listener.` `* @author Ken Weiner, kweiner@unicon.net` `* @version $Revision$` `*/`	Ohloh Rating: **0.0 / 5.0** Project ID: **3513** Lines of code: **29** File size: **667 Byte**
Project: **JA-SIG Sandbox**	**IListener** in *org.jasig.portal.portlet.servlet*	Lucene score: **1.4168**
	`package org.jasig.portal.portlet.servlet;`	Ohloh Rating:

Figure 5.3: Screenshot of the current version of *JBender* used for research and development. The areas in green (right column, metric chooser beneath search area) are mainly for research and debugging purposes and would not be implemented in a final application.

name and code snippet with highlighted occurrences of the search term. Furthermore it shows the entities name, implemented interfaces/extended superclass and containing package. On the left hand side of the result is the trustability information of the result's originating project: number of users, number of developers, main license of the project and of course the metric assigned trust value. Here the raw trust measurement is displayed as a floating point number to the user. In a second version of *JBender* we included a ranked assessment that maps the trustability to a scale from 0 to 10 to improve usability. (See Subsection 7.1.4 for more information on the trustability rankings.)

Figure 5.3 shows a screenshot of the current version of *JBender* we used for research and development. At the top there is a familiar search mask for text search, on the bottom are results for the executed query. The areas in green are mainly for research and debugging purposes and would not be implemented in a final application.

Chapter 6

RBender: Structured Search over Dynamic Code

This chapter will discuss our work in creating a code search engine with the goal of providing structured code search over source code of a dynamically typed programming language. We found that dynamic languages can—and should—be supported by code search engines via structured code search.

We see *RBender* as a first step to evaluate the feasibility of structured code search over dynamic languages. Due to a shift in the focus of this thesis the project *RBender* was put aside and never reached a running level.

6.1 Motivation

To our knowledge ***RBender*** is one of the first code search engines to actively support a dynamic programming language: Ruby. Though there exist several code search engines that have indices of Ruby source code, *e.g.*, Koders[1] and Google Code Search[2], none of those CSEs actively support to search within the structure of the Ruby source code.

With *RBender* we tried to address this shortcoming: the aim of *RBender* was to provide structured code search (see 3.2.2) through a code base of Ruby source code.

6.2 Based on Bender / JBender

The *RBender* prototype is based on the same *Bender* core as the *JBender* prototype. The parsing and analysing steps in the index creation chain were adapted to process Ruby source code. Upon reaching a running version of *RBender* we

[1] `http://www.koders.com`

[2] `http://www.google.com/codesearch`

planned to create a special GUI to allow for the structured functionality based search we were pursuing with *RBender*.

6.3 Code Search over Dynamically Typed Languages

One of the main differences of dynamically to statically typed languages is the lack of type annotations in the source code. This is an information shortfall that can hardly be compensated. Type information of static languages presents very valuable information that can be used for structured code search.[3] When trying to implement structured code search over dynamic languages one does not have access to this information.

Nonetheless, there is still a very clear structure woven through source code: although type information is missing, there still is structural information available for classes, methods and modules.

6.3.1 Parsing Ruby source code

In order to get the information for structured code search we performed a static analysis of the Ruby source code. For this the Ruby source code was parsed and an AST (Abstract Syntax Tree) was created. This syntax tree was then processed and the structural information about the code was extracted. To parse the Ruby source code we looked at the parser contained in the JRuby project[4]. This parser was used and expanded by at least two different projects: JRubyparser by Thomas Enebo[5] and RDT by Sommerlad et al. [6]. We based our parsing of the Ruby source code on the work of these two projects, concerning in particular the matter of Ruby comments in the AST.

Searchable fields

When parsing the Ruby AST we wanted to create a logical model in the code search engine's index that provides a fine granularity in order to let the developer search for code at a very low level, *i.e.*, single classes or methods. To allow for this and to account for the dynamic form of Ruby source code we decided to let the user search for logical entities of the following kinds:

```
File, Module, Class, Method
```

For each entity that is parsed from the AST and saved in the index various fields are searchable. Lucene supports logic searches over multiple fields so that multiple specifications for fields can be combined. Some examples for searchable fields are:

[3]More on structured code search in 3.2.2.

[4]A pure Java implementation of the Ruby programming language — `http://jruby.org`

[5]JRubyparser — `http://kenai.com/projects/jruby-parser`

[6]Refactoring for Ruby Development Tool [8] — `http://r2.ifsoftware.ch/trac`

- the entity's name
- the whole source code
- the entity's attached comments
- the body of the entitiy
- the parent entities
- the entity's dependencies (`include`, `require`, `load`)

Static analysis of dynamic code

Currently dynamically typed languages like Ruby or Python still lack powerful static analysis tools as they exist for mainstream statically typed languages like Java or C++. This makes it a lot harder to provide structured code search.

The static analysis of Ruby source code we performed provides results upon which one could base structured code search. However there is of course still a lot of space for improvement. A better static analysis of Ruby code could for example work with type inference to compensate (partially) for the missing type information. Use of analysis tools that are in development for Ruby would possibly be of great help here. An example is DiamondBack Ruby[7], which provides type checking through type inference for Ruby 1.8 [1, 10].

In order to maximise the information extracted from dynamic source code like Ruby it would furthermore be advisable though to use some sort of flow analysis as well. One of the easy ways to achieve this would be by implementing unit testing in search queries. (See 3.2.2 for more information on unit testing as a sort of query formulation.) RSpec for example is a tool for unit testing in Ruby and an implementation thereof can promise results that are similar to the unit test approaches implemented for Java source indexes (*e.g.*, Merobase, S6).

6.4 Discussion of RBender

Parsing Ruby source code is a challenge for a developer. The subject of static or dynamic analysis of dynamically typed programming languages is its own field of research and we can not cover it in full extent. During the development of *RBender* several problems came up; there are two issues we would like to point out here.

6.4.1 Comments in Ruby

Source code comments constitute structural information that is very valuable for use by search engines. It is therefore desirable to collect all comments, if possible

[7]DiamondBack Ruby — `http://www.cs.umd.edu/projects/PL/druby/`

with their respective affiliation (*e.g.*, class or method). This task is not easy to tackle: in Ruby comments can appear almost everywhere in the source code and therefore at almost every possible level within the AST. Also there is no clear way to assign a comment to a specific logical element like a method or a class like there is for example with Javadoc in Java. Although there are conventions[8] those are not a firm base whereupon one can base interpretation for all Ruby comments. Comments written by different people may have different intent and therefore a different meaning.

6.4.2 Dynamic definitions

Ruby allows the user to make dynamic class or method definitions at runtime. Consider the following snippet of Ruby code:

```
if bool
  class Foo
    def bar
      puts 'Foo says hello!'
    end
  end
else
  class Foo
    def bar
      puts 'Foo says goodbye!'
    end
  end
end

Foo.new.bar
```

In the above shown example of a dynamic definition a class `Foo` is defined. The definition depends upon the runtime value of the variable `bool`. Consequently the method `bar` behaves differently depending on which definition was actually made. After a static evaluation it is not easily possible to predict what the call in the last line actually 'puts'. Such definitions are difficult to handle for a static analysis tool and therefore also for a search engine: does the class displayed in a search result really work as expected or does it possibly change its behaviour?

These two mentioned examples—erratic comments and dynamic definitions—show structures that are hard to accommodate: they call for a very flexible interpretation of the Ruby AST. The analysis of Ruby source code with the goal of enabling structured code search should be as sophisticated and fine-grained as possible. Only through this can the search engine use the full potential of the structural information provided by the source code and, eventually, provide a better search. The current implementation in *RBender* is simple but a first step in

[8]Example: RSpec[9]—a unit testing framework for Ruby—interprets comments preceding methods/classes directly as method/class comments respectively. A newline between the comment and the method/class breaks this link though.

[9]RSpec project: `http://rspec.info`

this direction. To further improve the extraction of information from the source code and therewith the structured search, it would be desirable to include some of the Ruby static analysis tools that are in development at the moment (for example Diamondback Ruby).

Chapter 7

Evaluation

7.1 Attributes of the proposed Trustability Function

This section discusses attributes of the trustability metric we introduced in Chapter 4. We are discussing the metric function coupled with the raw data derived from Ohloh. Therefore we need to specify certain terms more clearly: as described in Chapter 5 we used the "stacked users" from Ohloh as our interpretation of user votes v_p in the calculation of T_p. We therefore use the two terms users and user votes synonymous in this chapter.

7.1.1 Stability

The trustability function we proposed provides a metric that is stable over time *i.e.*, over updates of the source code and the metadata sets concerning the source code.[1] Among other things, this is due to the fact that we calculated trustability on a per-project basis; this greatly increases the amount of information that goes into the calculation of the trustability for a single search result. A bigger amount of data allows for changes in the project's developing team, changes in user numbers and changes in the portfolio of involved developers without fundamentally changing the trust value calculated by the proposed metric.

Stability is an important attribute a metric must offer for code search users. As the trustability function provides the user with an abstract number, the user will need some time to get accustomed to it. It is then of course vital that the significance of that number stay the same over time and does not change too abruptly without good reason.

[1]This does not mean that trustability of a project does not change at all. The trustability of a project does for example change when new developers with high karma start contributing to it.

7.1.2 Power law distribution

We found that our input data (*i.e.*, the user-generated data that we crawled from Ohloh) follows a power law distribution: these are the number of votes per project ($r = 0.95157$), the number of commits per developer per project ($r = 0.89207$), as well as the number of projects per developer ($r = 0.85029$). Therefore we applied *log* and *tf-idf* weighting so that the trustability metric is not dominated by high values. At the moment project trust values calculated by the trustability function T_p range from zero to about 52, developer karma ranges from zero to about 72. In Section C.1 the reader can find graphs illustrating the distributions of the raw data we collected from Ohloh. In Section C.2 there are graphs showing the distribution of karma and trustability values we calculated based on the raw data from Ohloh.

7.1.3 Some top results

Table 7.1 illustrates the top ten results from the project ranking by our trustability metric. These are the 'most trustable' projects according to our metric. The trustability of a project does not simply correspond directly to its number of users (votes). 'grepWin' for example has only 32 user votes on Ohloh but is ranked by us with top trustability. This is because the developers of grepWin are very active in other projects and have a high karma value thus making the project trustable.

Top projects (by trustability)	**trust value (T_p)**	**no. of votes (v_p)**
grepWin	51.6	32
GNU Diff Utilities	51.18	645
Eclipse Ant Plugin	49.76	136
Eclipse Java Development Tools	48.36	647
Crimson	42.41	2
GNU binutils	42.18	525
syrep	42.12	2
GNU M4	41.85	54
gzip	41.61	261
Forgotten Edge OpenZIS	40.86	1

Table 7.1: Top projects by trustability. Also indicated is the number of votes of the project.

Table 7.2 shows the top ten developers by karma. Manual verification determined those are all developers who 'deserve' high karma rank: they have high numbers of commits to many very successful open source projects and are generally involved in their development for several years. The user 'darins' for example is one of the key developers for the Eclipse Project. His involvement is registered since 2001 with about 8000 commits for the core project and thousands of more commits to popular Eclipse plugins like JDT or Eclipse Ant. 'amodra' is a developer working with major contributions at GNU binutils and the Gnu Compiler

Collection.

Top developers (by karma)	**karma value (K_d)**
darins	71.97
amodra	70.11
darin	69.09
nickc	67.14
Dani Megert	66.51
mlaurent	66.14
Paul Eggert	65.89
kazu	65.78
rth	65.25
hjl	65.04

Table 7.2: Top developers by karma. (See Section 4.1)

7.1.4 Ranking by trustability

The first trustability metric we implemented and tested on *JBender* was the plain trustability function presented in Chapter 4. This function is an unbounded continuous metric. Figure 7.1 shows the distribution of the trustability values.

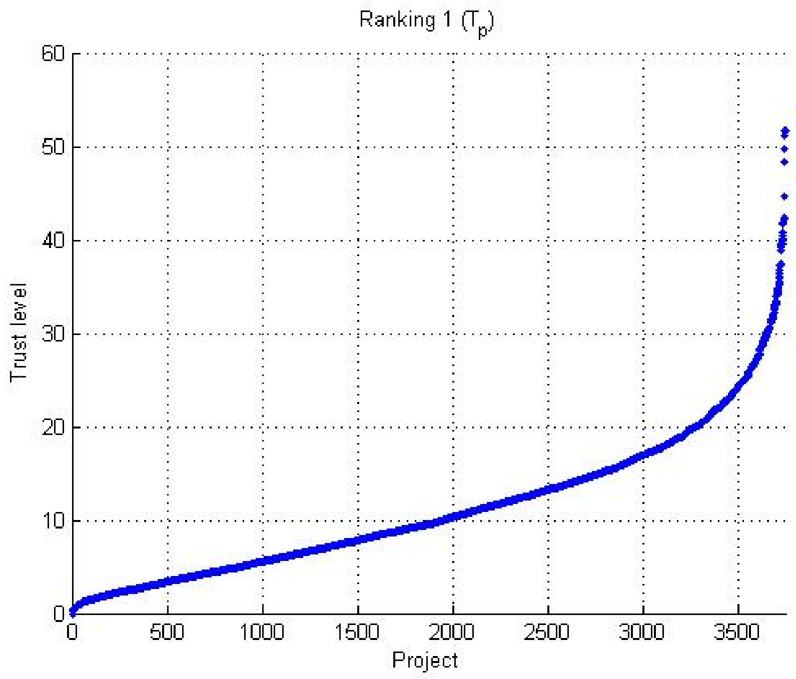


Figure 7.1: Ranking 1, representing the continuous trust values calculated by T_p. We can read from this graph that about 1900 out of 3700 projects have a trust value lower than 10.

After conducting a small selective survey we quickly observed that the bare numeric values assigned by our trustability function leave the user at a loss about their meaning. Although the user could be provided with the present range in which the trust values lie,[2] this range is very likely to change with updates of project meta data. Table 7.1 shows the top ten projects by trustability. If one considers only the indicated trust values T_p it is not intuitively clear if those are trustable projects or not. Concerning the fact that the table is showing the top ten projects this should be the case!

This calls for a trustability ranking that is stable over time—*i.e.*, over metadata updates, addition of further projects or similar changes—and that gives the user an easier way to compare search results against each other.

Discrete ranking

We decided to use a discrete ranking that is a strict weak ordering on the range from zero to ten: each project gets assigned a trustability level from zero to ten, zero meaning not trustable (*i.e.*, no trustability related information is available), ten meaning the project is very trustable (*i.e.*, a large number of good and active developers worked on the project).

This presents the user with a far clearer picture on how to classify a project when presented the trust value: the examples from Table 7.1 get assigned trustability rank 10 (being the top ten results from about 3700). On the given scale from one to ten, this obviously means that the projects are quite trustable.

The current version of the *JBender* prototype has access to three different kinds of project trustability rankings:

- Ranking 1 is the original ranking calculated by the trust metric T_p.
- Ranking 2 is a discrete ranking from 0 to 10 whose levels are uniformly distributed over the 3700 projects, *i.e.*, with about 336 projects per level.[3]
- Ranking 3 is a discrete ranking where the calculated trust value T_p is normalized to the range 0–10.

The second listed ranking is problematic because there are equal numbers of projects for each level of trustability (0 through 10). This probably does not represent the user's understanding of how project trustability should be distributed: a distribution where the very trustable (10) projects appear less likely than the moderately trustable ones (5) seems more intuitive. Figure 7.2 shows a graph of the trust level distribution over all projects (sorted by trustability) for ranking 2.

The third listed ranking meets this above-mentioned concern: the discrete values are distributed as the original values by T_p, therefore the higher levels of trustability are less likely to occur. This however has the disadvantage that only very few projects out of the 3700 get assigned high levels: only 55 projects reach level 7 or higher, the majority lies within the range of 1–3. Figure 7.3 shows a graph of

[2]At the time of writing project trustability ranges from 0 to about 51.

[3]3700 projects, 11 levels (0 through 10) $\rightarrow 3700/11 \approx 336$ projects per level.

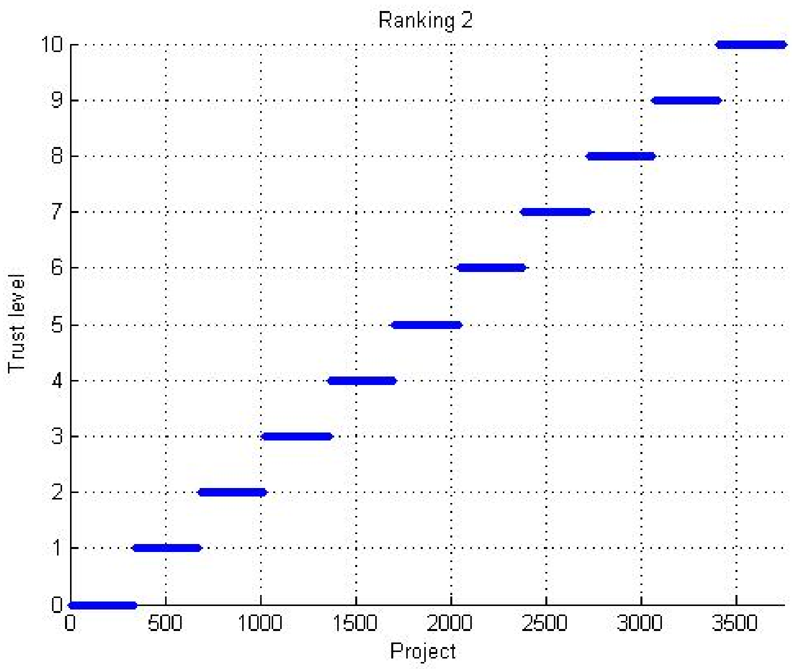


Figure 7.2: Ranking 2, uniformly distributed trust levels. On each level there are equal numbers of projects.

the trust level distribution over all projects (sorted by trustability) for ranking 3.

Presently the user of *JBender* can choose which ranking should be applied for a search, *i.e.*, what the distribution of the trustability values should look like. This option to choose the ranking is a choice that we would not want to impose upon the user in a final search engine. We put it in *JBender* though because this is a choice that we have not finally made ourselves. The projection of the calculated values T_p to the levels 0 to 10 is directly responsible for the distribution of the trustability values finally presented to the user. Figure 7.4 shows the three rankings in one graph for comparison. It is desirable that the distribution of the values presented corresponds with the user's intuitive understanding of how trustability of projects is distributed. This topic needs further research to study how this distribution should look like. Only after this a decision upon the sort of distribution of the trustability values should be made.

7.2 A Comment on Ohloh's Rankings

Our implementation of the introduced trustability metric in *JBender* is based on metadata crawled from Ohloh. As Ohloh already provides several rankings of its own it is nessesary that we discuss these here. We also explain why we did not use Ohloh's rankings directly to assess code trustability.

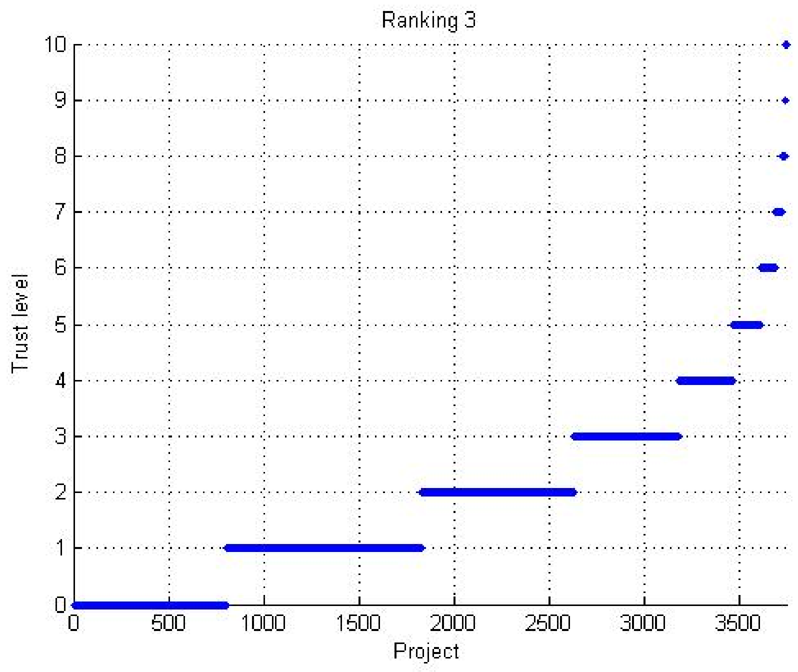


Figure 7.3: Ranking 3, discrete trust levels normalized to the range 0–10. More than 90% of projects have a trust level of 4 or below.

7.2.1 Developer ranking

The Ohloh website provides its own measurement of developer "karma", called *kudo-rank*. Kudo-ranks are based on a mix of user votes for projects and of user votes for developers, called *kudos*. The exact calculation is not disclosed by Ohloh, though one can extract some information from their homepage.[4] Developers get a high kudo rank through

- getting kudos from other developers on Ohloh, and
- through commits they made to successful projects listed on Ohloh.

Successful here means the project got "stacked" by users many times. This corresponds to a degree with our karma calculation proposed in Section 4.1.[5] However user participation for kudos is very low on Ohloh and therefore weighted strongly. As a consequence, a small clique of developers can vote itself up to top kudo ranks, independent of their contributions to any project. Therefore, we decided against including kudo-ranks from Ohloh in our trustability function.

[4] `http://www.ohloh.net/about/kudos`

[5] "Stacked" is Ohloh's term for a user adding himself as an active user of a project. In our implementation of the trustability metric in Chapter 4 this is interpreted as a user vote.

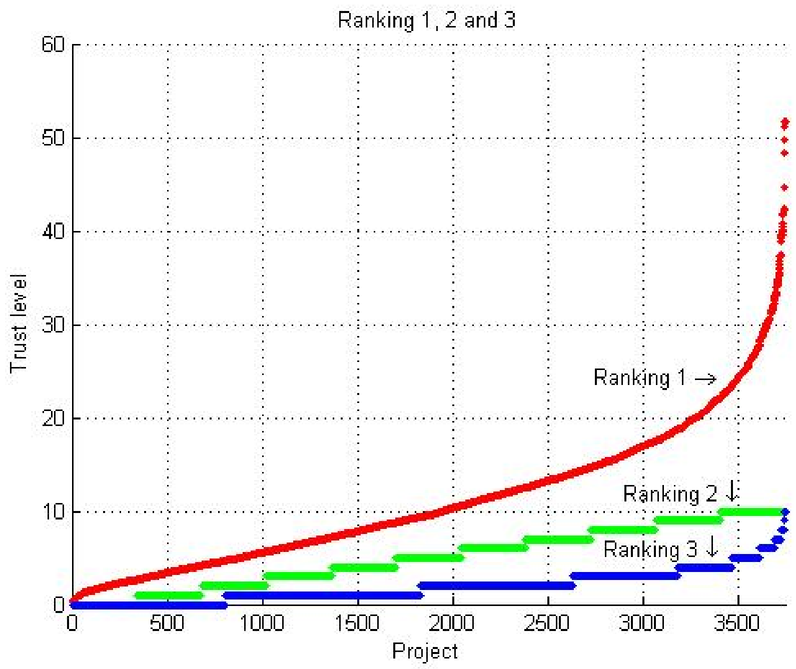


Figure 7.4: Ranking 1 (continuous), 2 and 3 (discrete) for comparison.

7.2.2 Project ranking

Ohloh also provides a ranking of projects for its search mechanism. By default projects are sorted by user votes. Table 7.3 shows the top ten projects listed by Ohloh. One can notice that those are all very popular and successful open source projects. Basically this could be understood as a very first simple assessment of trustability. It does not however take into account cross-project activity of developers and is therefore less accurate and easier to tamper with. Also the pure success of a single project must not necessarily correspond with good and trustable source code of this project.[6]

7.3 JBender Prototype

In this section the *JBender* prototype and some of its features are discussed. *JBender* was designed as a proof-of-concept prototype. We used it to implement the trustability metric we proposed and to study first results. If *JBender* were to become a productive project the performance and the size of the codebase and metadatabase would have to be improved.

[6]In Ohloh's defense we want to add that the ranking of projects never was intended for the purposes of the proposed trustability ranking.

Top projects (by votes)	no. of votes (v_p)
firefox	7207
subversion	5687
apache	5107
mysql	4834
php	4081
openoffice	3118
firebug	3109
gcc	2586
putty	2519
phpmyadmin	2412

Table 7.3: Top projects by votes as ranked by Ohloh. (As of 2009-12)

7.3.1 Evaluation in user study

We see the *JBender* prototype as a first step into the right direction: to take into account human and social factors when deciding upon the value of code search results. We have not conducted a user study to evaluate *JBender*'s contribution. For one, the tool was not designed to be in actual productive use. It would take some time to 'round off the edges' and really make *JBender* fast and easily usable to the broad public. Secondly, designing the user study would itself demand careful preparation: How does one measure the user's trust in the results he is presented with? How does one measure the advantage a user gets when he is provided with additional trustability information? These would be interesting topics for further research.

7.3.2 Limitations of per-project information gathering

The metadata we collect for *JBender*'s metadatabase is collected on a per-project basis. For example, we collect the description of projects, the URL of the project homepage and which developers worked on a project. Collecting data in this way has both advantages and disadvantages compared to data collection on a per-file basis.

Collecting metadata for whole projects instead of single files allows for better information density when seen from a trustability point of view. A lot of the data we used to assess result trustability was manually specified by users of Ohloh or deduced from such data. Relying on this data is possible for whole projects. It is not sensible though to expect similar user contributions for single files of source code. Consequently, comparing user votes or ratings would hardly be possible on a per-file basis. Moreover, the calculation of a trustability metric on a single-file basis would have to pay special attention in order not to be too dependent on single outstanding developers. To correctly measure the impact of the few different developers on certain areas of source code and to compare it with their overall trustworthiness ("karma"), one would have to dive deep into repository mining topics. This could be a promising field for future work but goes beyond the scope of this work.

On the other hand there is information that should be collected for each file. First and foremost this would be the license under which a specific file was published. *JBender* has at this point only knowledge of the license(s) used for a whole source project. This can pose a problem when a project uses more than one license for seperate files/sub-projects: Assigning the correct license to a certain file is not easy. It is not trivial for a CSE, to get licenses on a file-basis. License headers within files are only a convention and may be missing sometimes or licenses may be specified ambiguously for projects within projects *etc.*.

7.3.3 Performance

Performance is a key attribute for search engines. *JBender*, though, is only a proof-of-concept prototype, therefore we did not pursue this subject. The search in the source code index of *JBender* via the *Bender* core is reasonably fast: to search through more than 200.000 source files *JBender* takes about 100–200 milliseconds. The access to the metadata on the other hand is rather slow. Metadata is saved in the form of *.json* files which are parsed at run-time. This results in access times of around 800 milliseconds. This value depends linearly on the number of developers and users within *JBender*'s metadatabase. In a productive version the performance could be improved by storing the metadata in a fast relational database.

Chapter 8

Conclusion

In this thesis we have presented an approach to improve the *trustability* of search results. Trustability of search results is important, for developers to quickly assess search results from external code bases before integrating them into their local code base.

Our aim is that the developer gains a quick estimate of code trustability in his results. We therefore created a trustability function, that calculates a trust value for each originating project depending on the project's meta data. This trust value is an indication of the project's quality, its popularity and the quality of its source code. Upon reaching a sufficient size of source code and metadata index, it would also be feasible to sort search results according to their trust value. As result relevance is paramount, the trustability metric would then be used to choose from a pool of *relevant* search results. Under the premise that all results comply with the user's technical specification, this would provide the user working results of the best quality.

We have proposed T_p as a *trustability metric* for software projects. The proposed metric uses collaborative filtering of user votes and cross-project involvement of developers. The trustability of projects is assessed and each project is assigned a trust value.

We also have presented *JBender*, a proof-of-concept prototype code search engine that implements the introduced trustability metric. It allows developers to quickly assess the trustability of search results presented to them. We have discussed the choice of our trustability metric and presented preliminary results from a first evaluation.

In this thesis we also discussed findings from our work on a second prototype, *RBender*. *RBender* is a search engine that parses a dynamically typed programming language—Ruby—and provides a structured search over it. The *RBender* project was a first evaluation of the feasibility of structured search on dynamic code—it did not reach a running level.

8.1 What is missing?

JBender was designed as a proof-of-concept prototype—for it to be a useful productive tool some changes and improvements would have to be made. The source code and metadata base of *JBender* should be increased by crawling more data from Ohloh and possibly other sources. The performance of *JBender* should be improved to provide the user a fast and easy to use search engine. Improvement of performance would mainly be important in the web front end and for the storage (and access) of the metadata. A final decision would have to be made upon the trustability ranking: which of the presented rankings is to be applied and how is it presented to the user? Furthermore the usability of the structured code search by *JBender* should be improved to catch up with state-of-the-art code search engines.

8.2 Future Work

The current trustability metric is defined per project because the trustability data is collected on a per-project base. We would like to combine this data with file-base data, *e.g.*, with code ownership data from project history. This would make it possible to assess the trustability of single classes (or even methods) based on developer's karma.

JBender is our first cut on the subject of trustability enhanced code search engines. It would be interesting to conduct an extensive user study to see how users work when provided with trustability data and to see if this makes their searches more efficient.

It would also be of interest to do empirical studies on what mental model users have of the trustability of source code. What makes source code trustable, what makes it untrustable? How is trustability distributed over many files/projects?

We would also like to compare the proposed trustability metric with other trustability measurements. It might also be promising to combine the proposed trustability metric, which is currently based on human factors only, with technical trustability assessments such as, *e.g.*, test coverage.

RBender, a code search engine providing structured search over dynamic code, never saw the light of the day. Dynamically typed languages are suitable for structured search, they do however pose a challenge to code search engine developers and researchers. It might be promising to work with some of the dynamic and static analysis tools that are currently under development to increase the information gathered from dynamic code.

Appendix A

A.1 JBender's Source Code Index

A.1.1 Stored Fields (not indexed)

Field Name	Description
FILE_NAME	Name of the source file.
FILE_PATH	Path to the file (e.g. for download).
FILE_INDEX_TIMESTAMP	Time of indexing.
FILE_SIZE	Size of the file in byte.
FILE_LOC	Number of lines of code in the file.

Table A.1: Stored fields in *JBender*'s source code index. These are provided with results but cannot be searched.

A.1.2 Searchable Fields (indexed)

Field Name	Description
COMPLETE_SRC_CODE	The complete source code contained in a file.
COMPLETE_JAVADOC	The complete javadoc comment code in a file.
PACKAGE	The package information of the source file.
IMPORTS	All import statements of the source file.
INTERFACES	Any implemented interfaces in this source file.
SUPERCLASSES	All superclasses (from public and private classes) extended in this source file.
CLASS_NAMES_ALL	Names of all classes (public and private) in this source file.
METHOD_NAMES_ALL	All method names defined in this source file.
METHOD_BODIES_ALL	All method bodies defined in this source file.
METHOD_NAMES_PUBLIC	Names of all public methods.
METHOD_BODIES_PUBLIC	Bodies of all public methods.
METHOD_NAMES_PRIVATE	Names of all private methods.
METHOD_BODIES_PRIVATE	Bodies of all private methods.
METHOD_NAMES_PROTECTED	Names of all protected methods.
METHOD_BODIES_PROTECTED	Bodies of all protected methods.
METHOD_NAMES_STATIC	Names of all static methods.
METHOD_BODIES_STATIC	Bodies of all static methods.

Table A.2: Searchable fields in *JBender*'s source code index.

A.1.3 Top Term Tokens for respective Terms

	Complete Source Code		**Only javadoc comments**	
rank	*count*	*term*	*count*	*term*
1	167117	org	94711	org
2	141193	string	85625	copyright
3	120449	java	80681	http
4	113549	null	79810	implement
5	97434	util	79473	distribut
6	91164	c	79317	c
7	87530	http	78145	www
8	85751	copyright	75924	licens
9	80923	www	73093	right
10	80171	eclipse	72903	return
11	78333	object	72386	param
12	77658	v	72006	author
13	76195	license	71489	v
14	74144	param	71392	public
15	73882	TRUE	71102	reserv
16	73522	implementation	69030	term
17	71931	author	68949	initi
18	71869	rights	68442	avail
19	71207	reserved	67888	contributor
20	69552	available	67549	materi

Table A.3: Top term tokens in complete source/complete javadoc.

	All Class Names		**Inherited Superclasses**	
rank	*count*	*term*	*count*	*term*
1	738	Messages	7381	TestCase
2	481	Activator	1371	EObjectImpl
3	335	AllTests	1290	Exception
4	154	Constants	945	Action
5	107	Main	923	NLS
6	101	Util	848	ItemProviderAdapter
7	88	Node	631	BaseTestCase
8	82	ParseException	605	JPanel
9	71	Base64	568	RuntimeException
10	67	Handler	534	BasicCursorTestCase
11	66	Utils	520	Thread
12	64	Attribute	506	AbstractUIPlugin
13	63	BookCategory	473	BaseCase
14	61	Location	430	EventObject
15	55	User	380	Composite
16	54	Connection	374	AbstractAction
17	52	ResourceLoader	318	Dialog
18	52	Token	314	ServerBasePacket
19	50	Test	301	Plugin
20	49	Client	292	WizardPage

Table A.4: Top term tokens for class names/superclasses.

	Import Statements		**Implemented Interfaces**	
rank	*count*	*term*	*count*	*term*
1	902	java.io.Serializable	2014	Serializable
2	895	org.eclipse.osgi.util.NLS	1353	java.io.Serializable
3	606	org.eclipse.emf.ecore.EObject	1313	Runnable
4	598	junit.framework.TestCase	1287	IEditingDomainItemProvider
5	591	java.util.List	1236	EObject
6	542	java.util.*	450	Comparator
7	491	java.io.IOException	435	EventListener
8	425	junit.framework.Test junit.framework.TestSuite	414	Cloneable
9	387	java.io.*	400	XmlAnySimpleType
10	378	java.util.Map	345	BundleActivator
11	344	org.eclipse.emf.common.util.EList org.eclipse.emf.ecore.EObject	319	Comparable
12	318	java.util.EventListener	302	ActionListener
13	254	java.util.ArrayList java.util.List	242	EFactory
14	237	org.eclipse.emf.ecore.EFactory	241	EPackage
15	227	javax.xml.namespace.QName	233	Runnable Runnable
16	226	java.util.EventObject	222	Command
17	221	java.util.ArrayList	202	XMLEvent
18	219	java.util.Iterator	187	IWorkbenchPreferencePage
19	218	javax.xml.stream.XMLStreamReader	170	Enumerator
20	182	java.io.File	170	ITreeContentProvider

Table A.5: Top term tokens for imports/interfaces.

Appendix B

B.1 JBender's Metadata Base

B.1.1 Information stored per project

- The project's full name
- Description of original project
- Project homepage
- A list of descriptive tags assigned by Ohloh
- Ohloh's rating of the project
- Enlisted repositories
  - Type of repository (GIT, SVN, CSV, ...)
  - URL
  - Time of last access by Ohloh
  - Path within the URL
- Licenses
  - Exact type of license
  - No. of files with that license in the project

- Employed programming languages
  - Name of language
  - Total lines of code
  - No. of source lines
  - No. of comment lines
  - No. of blank lines
- List of all developers of the project
  - Name and ID of developer
  - Kudo rank of developer
  - No. of commits
  - Programming language of commit
- List of all users of the project
  - Name and ID of user
  - Kudo rank of user
  - Stack size

Appendix C

C.1 Raw Data from Ohloh

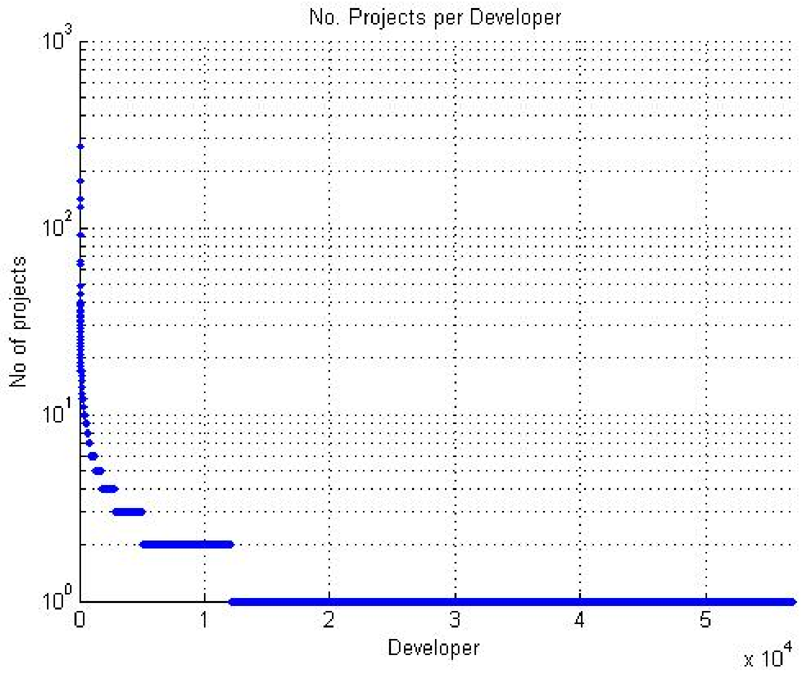


Figure C.1: Distribution of the number of projects developers are involved in, *i.e.*, have committed to.

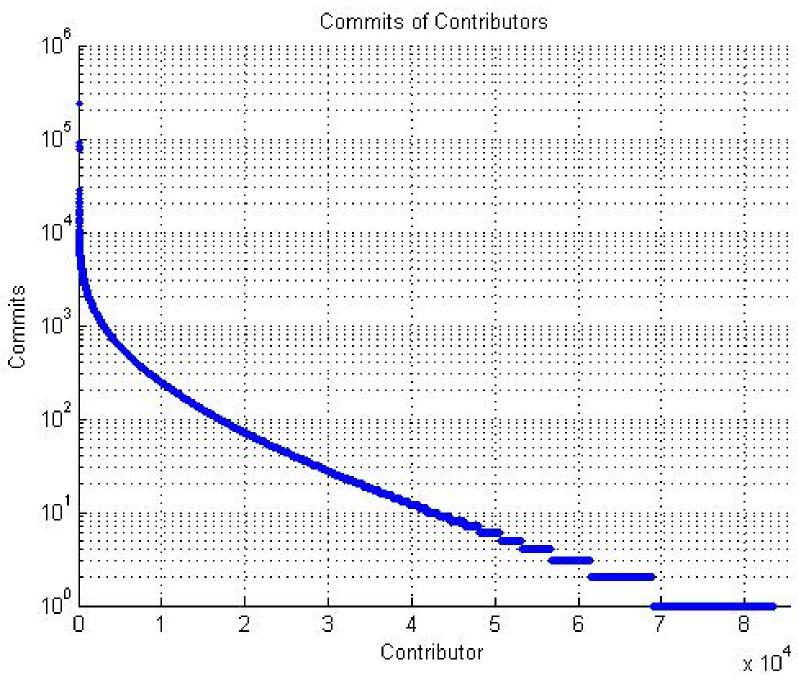


Figure C.2: Distribution of the number of commits per developer per project.

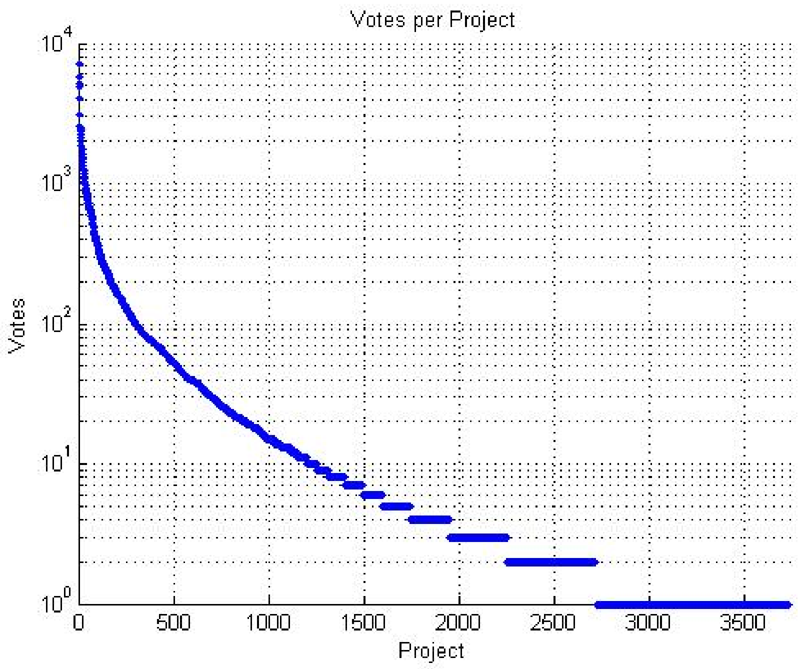


Figure C.3: Distribution of the number of user votes per project.

C.2 Metadata based on Ohloh

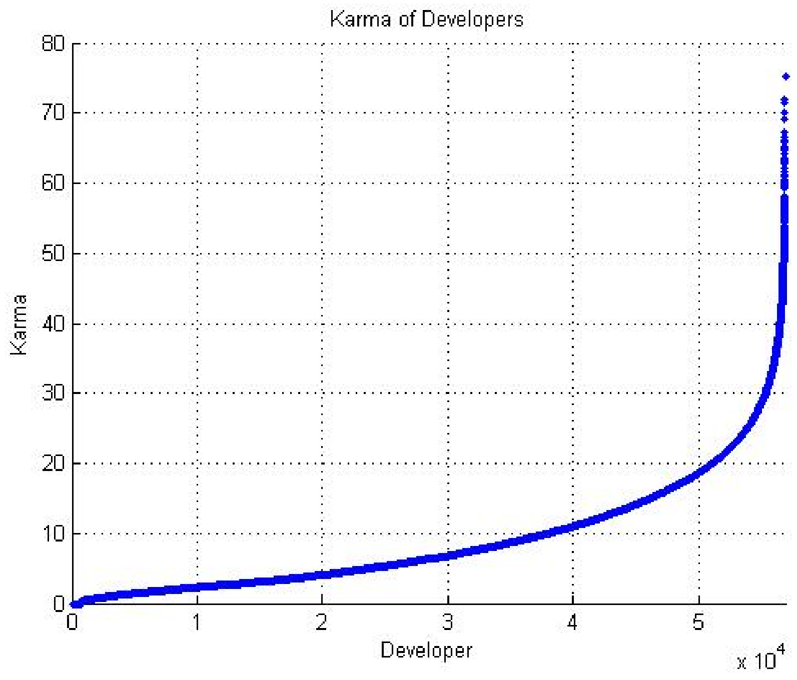


Figure C.4: Distribution of karma values K_d of developers.

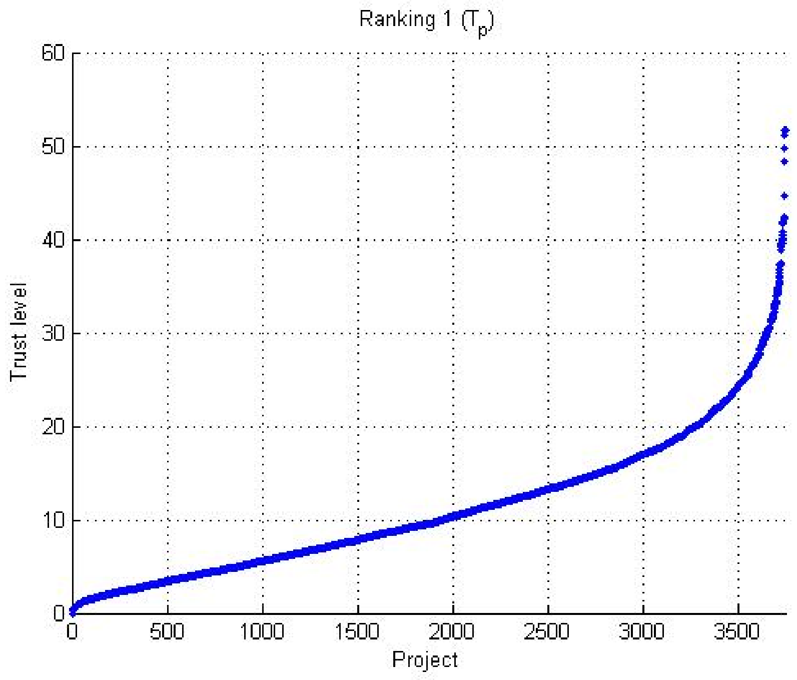


Figure C.5: Distribution of trustability values T_p of projects.

List of Figures

List of Tables

Bibliography

[1] Jong-hoon D. An, Avik Chaudhuri, and Jeffrey S. Foster. Static typing for Ruby on Rails. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering. Auckland, New Zealand. November 2009*. IEEE/ACM, 2009.

[2] Oliver Arafat and Dirk Riehle. The comment density of open source software code. *In Companion to Proceedings of the 31st International Conference on Software Engineering (ICSE 2009)*, pages 195–198, 2009.

[3] Oliver Arafat and Dirk Riehle. The commit size distribution of open source software. *Hawaii International Conference on System Sciences*, 0:1–8, 2009.

[4] Sushil Bajracharya, Adrian Kuhn, and Yunwen Ye. Suite 2009: First international workshop on search-driven development - users, infrastructure, tools and evaluation. In *Software Engineering - Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on*, pages 445–446, 2009.

[5] Sushil Bajracharya, Trung Ngo, Erik Linstead, Yimeng Dou, Paul Rigor, Pierre Baldi, and Cristina Lopes. Sourcerer: a search engine for open source code supporting structure-based search. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 681–682, New York, NY, USA, 2006. ACM.

[6] Albert-Laszlo Barabasi. *Linked: How Everything Is Connected to Everything Else and What It Means*. Plume, reissue edition, April 2003.

[7] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley, 2003.

[8] Thomas Corbat, Lukas Felber, Mirko Stocker, and Peter Sommerlad. Ruby refactoring plug-in for eclipse. In *OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 779–780, New York, NY, USA, 2007. ACM.

[9] Mark Driver. Dynamic programming languages will be critical to the success of many next-generation AD efforts. Technical report, Gartner Inc., ID G00163617, December 2008.

[10] Michael Furr, Jong-hoon D. An, Jeffrey S. Foster, and Michael Hicks. Static type inference for Ruby. In *Proceedings of the 24th Annual ACM Symposium on Applied Computing, OOPS track. Honolulu, HI. March 2009*, 2009.

[11] Rosalva E. Gallardo-Valencia and Susan E. Sim. Internet-scale code search. In *Search-Driven Development-Users, Infrastructure, Tools and Evaluation, 2009. SUITE '09. ICSE Workshop on*, pages 49–52, 2009.

[12] Vinicius C. Garcia, Eduardo S. de Almeida, Liana B. Lisboa, Alexandre C. Martins, Silvio R. L. Meira, Daniel Lucredio, and Renata P. de M. Fortes. Toward a code search engine based on the state-of-art and practice. In *Software Engineering Conference, 2006. APSEC 2006. 13th Asia Pacific*, pages 61–70, 2006.

[13] David A. Grossman and Ophir Frieder. *Information Retrieval: Algorithms and Heuristics*. Kluwer Academic Publishers, Norwell, MA, USA, 1998.

[14] Florian S. Gysin. Improved social trustability of code search results. In *Proceedings International Conference on Software Engineering, ICSE '10, Student Research Competition*, 2010. To appear.

[15] Florian S. Gysin and Adrian Kuhn. A trustability metric for code search based on developer karma. In *ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation, 2010. SUITE '10.*, 2010. To appear.

[16] Marti A. Hearst. *Search User Interfaces*. Cambridge University Press, 1 edition, September 2009.

[17] Reid Holmes and David Notkin. Enhancing static source code search with dynamic data. In *Search-Driven Development-Users, Infrastructure, Tools and Evaluation, 2010. SUITE '10. ICSE Workshop on*, 2010.

[18] Oliver Hummel, Werner Janjic, and Colin Atkinson. Code Conjurer: Pulling reusable software out of thin air. *Software, IEEE*, 25(5):45–52, 2008.

[19] Makoto Ichii, Yasuhiro Hayase, Reishi Yokomori, Tetsuo Yamamoto, and Katsuro Inoue. Software component recommendation using collaborative filtering. In *Search-Driven Development-Users, Infrastructure, Tools and Evaluation, 2009. SUITE '09. ICSE Workshop on*, pages 17–20, 2009.

[20] Katsuro Inoue, Reishi Yokomori, Tetsuo Yamamoto, Makoto Matsushita, and Shinji Kusumoto. Ranking significance of software components based on use relations. *Software Engineering, IEEE Transactions on*, 31(3):213–225, April 2005.

[21] Werner Janjic, Dietmar Stoll, Philipp Bostan, and Colin Atkinson. Lowering the barrier to reuse through test-driven search. In *Search-Driven Development-Users, Infrastructure, Tools and Evaluation, 2009. SUITE '09. ICSE Workshop on*, pages 21–24, 2009.

[22] Evangelos Katsamakas and Nicholas Georgantzas. Why most open source development projects do not succeed? In *ICSEW '07: Proceedings of the 29th International Conference on Software Engineering Workshops*, pages 123+, Washington, DC, USA, 2007. IEEE Computer Society.

[23] Adrian Kuhn. Automatic labeling of software components and their evolution using log-likelihood ratio of word frequencies in source code. In *MSR*

'09: Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories, pages 175–178. IEEE, 2009.

[24] Otávio Augusto Lazzarini Lemos, Sushil Bajracharya, Joel Ossher, Paulo C. Masiero, and Cristina Lopes. Applying test-driven code search to the reuse of auxiliary functionality. In *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing*, pages 476–482, New York, NY, USA, 2009. ACM.

[25] Otávio Augusto Lazzarini Lemos, Sushil K. Bajracharya, Joel Ossher, Ricardo S. Morla, Paulo C. Masiero, Pierre Baldi, and Cristina V. Lopes. Codegenie: using test-cases to search and reuse source code. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 525–526, New York, NY, USA, 2007. ACM.

[26] Masao Ohira, Naoki Ohsugi, Tetsuya Ohoka, and Ken I. Matsumoto. Accelerating cross-project knowledge collaboration using collaborative filtering and social networks. In *MSR '05: Proceedings of the 2005 international workshop on Mining software repositories*, volume 30, pages 1–5, New York, NY, USA, July 2005. ACM.

[27] Anna Patterson. Why writing your own search engine is hard. *Queue*, 2(2):48–53, 2004.

[28] Steven P. Reiss. Semantics-based code search. *Software Engineering, International Conference on*, 0:243–253, 2009.

[29] Steven P. Reiss. Specifying what to search for. In *Search-Driven Development-Users, Infrastructure, Tools and Evaluation, 2009. SUITE '09. ICSE Workshop on*, pages 41–44, 2009.

[30] David Röthlisberger, Marcel Härry, Alex Villazón, Danilo Ansaloni, Walter Binder, Oscar Nierstrasz, and Philippe Moret. Augmenting static source views in IDEs with dynamic metrics. In *Proceedings of the 25th International Conference on Software Maintenance (ICSM 2009)*, pages 253–262, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[31] Susan E. Sim, Charles L. A. Clarke, and Richard C. Holt. Archetypal source code searches: A survey of software developers and maintainers. In *IWPC '98: Proceedings of the 6th International Workshop on Program Comprehension*, pages 180+, Washington, DC, USA, 1998. IEEE Computer Society.

[32] Janice Singer, Timothy Lethbridge, Norman Vinson, and Nicolas Anquetil. An examination of software engineering work practices. In *CASCON '97: Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research*, pages 21+. IBM Press, 1997.

[33] Jamie Starke, Chris Luce, and Jonathan Sillito. Working with search results. In *Search-Driven Development-Users, Infrastructure, Tools and Evaluation, 2009. SUITE '09. ICSE Workshop on*, pages 53–56, 2009.