Exploiting Crowd Knowledge in the IDE

Master's Thesis submitted to the
Faculty of Informatics of the *Università della Svizzera Italiana*
in partial fulfillment of the requirements for the degree of
Master of Science in Informatics
Major in Software Design

presented by
Luca Ponzanelli

under the supervision of
Prof. Dr. Michele Lanza
co-supervised by
Alberto Bacchelli

June 2012

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Luca Ponzanelli
Lugano, 22 June 2012

To my father, my family and whoever backed me up to achieve this goal

"Any intelligent fool can make things bigger, more complex, and more violent. It takes a touch of genius – and a lot of courage – to move in the opposite direction."

Albert Einstein (1879-1955)

Abstract

Software development process is a non trivial endeavor. Documentation does not often hold the pace of change of the project, thus lowering the support that developers need to understand the system. To overcome this lack, developers consult other programmers with the hope of getting hints to overcome a problem they encountered, or to gather suggestions about design ideas they have. The research is often extended to online resources, such as tutorials and messaging boards. This practice is defined as *crowdsourcing*.

It is not surprising that nowadays there are online communities in which developers collaborate to solve problems and programming issues or to discuss design ideas. Among the many resources available on the web, Questions & Answer (Q&A) services are gaining popularity (*e.g., stackoverflow.com, Yahoo! Answers etc.*) and crowdsourcing is becoming an usual practice.

Even though the usage of Q&A services has dramatically increased, this new important resource has been scarcely taken advantage of by any Integrated Development Environment (IDE). Interacting with those communities requires developers to continuously switch between the IDE and the web browser to read the discussions and to then perform modification to the code, thus leading to interruptions in the programming flow that lowers the developers' performance.

In this thesis, we present Seahawk, an Eclipse plugin to integrate Stack Overflow crowd knowledge in the IDE. Seahawk allows developers to seamlessly retrieve Q&A from Stack Overflow in the IDE, link relevant discussions to any source code in a collaborative fashion by also attaching explanative comments, and to automatically generate queries from code entities.

Acknowledgements

This thesis could not have been possible without Prof. Dr. Michele Lanza, who did not just supervise me, but he pushed me to put the best effort in my work. Without the challenges of this thesis I could not have tasted the preliminary experience of the academic research. Thank you.

I must acknowledge Alberto Bacchelli for the patience in bearing with me for almost six months, and for the help and the experience he provided me, disregarding the time zones.

I would like to acknowledge the REVEAL group, including Dr. Marco D'ambros and Fernando Olivero, for the discussions and the exchange of ideas that took place in the weekly meetings.

A special acknowledgement is due to Remo Lemma, Patrick Zulian and Teseo Schneider for their support during the past two years, and to have become, somehow, companions of a long travel during my academic experience in Lugano.

I would like to thank and extend my heartfelt gratitude to my friends in Italy. During this two years, they have been a stable landmark to rely on. Especially, I would like to acknowledge Adama Faye for having tried to teach me english. Even though the results achieved by his student are still debatable, moving to Lugano would have been a harder experience without his help.

Last but not least, I would like to thank all the members of my family. Especially, my father Ernesto Ponzanelli, my mother Gabriella Bianchi, and my brother Claudio Ponzanelli. This achievement has been also possible because of you, for having encouraged me in trying this experience, for having provided me with all the support needed and for the comprehension you had during the hardest time. You are unique. Thank you.

Contents

Figures

Tables

Chapter 1

Introduction

Software engineering is a continuously evolving field. In its history, many aspects of this discipline have changed, thus requiring developers and engineers to hold the pace of change. Of course, the evolution of technology has played, and it is still playing, a prominent role in the software development process. Developers are continuously introduced to new technologies, components, ideas [RWZ10] and they often deal with systems depending on external libraries or even legacy code. The time they spend in maintaining such systems requires a good comprehension of the behavior of the system behaves and what the code is supposed to do. Researches have shown how developers spend 50% of their time in program comprehension tasks [LS81]. Moreover, Corbi *et al.* pointed out how "Software renewal tools are needed to reduce the costs of modifying and maintaining large programming systems, to improve our understanding of programs" [Cor89]. The employment of technology in assisting code comprehension has been tackled by research, and industry, in the last two decades by producing tools to support developers in program comprehension. For example, Kuhn *et al.* take advantage of software cartography and information retrieval techniques to represent a program as a topic-map [KELN10] , Sridhara *et al.* automated the creation of Java comment from code analysis [SHM+10], and Haiduc *et al.* presented a system to generate textual summaries from source code [HAMM10][HAM10].

Among the tools and the systems developers have at their disposal, recommendation systems are available as well. A recommendation system provides suggestions to developers by helping them face and navigate the huge amount of information space a project has. According to Robillard *et al.*, "the recommendation system for software engineering (RSSE) are emerging to assist developers in various activities, from reusing code to writing effective bug reports" [RWZ10]. RSSEs could be a potential means to help developer in code comprehension tasks, or to assist them to retrieve additional information while developing or modifying part of a system.

Apart from the technological support in code understanding, developers also need to modify or to develop new components. Although the software development process outcome is heavily based on the knowledge and the creativity of software developers [Ye06], writing code often requires knowledge beyond that which developers already possess [KDV07]. Developers create and maintain software systems by standing on the shoulders of many others, by asking teammates for advice as well as by reusing components, consulting project documentation, books and manuals, or referring to online resources and tools [STvDC10]. However, project documentation is commonly inadequate [LVD06], manuals tend to be outdated, and books may be hard to retrieve or link to the actual task. Moreover, documentation is often biased by the technical aspects of the solution described,

missing information regarding the context of the design process [HP00]. For these reasons, often developers' knowledge needs can only be satisfied by posing questions to other programmers or teammates [KDV07][HP00]. With the aim of leveraging these circumstances and the success of social media, Question and Answers (Q&A) online services offer infrastructures to support knowledge exchange between programmers. The reciprocal collaboration applies for every kind of experience level a developer has, ranging from students to experienced developers.Even though many studies (*e.g.,* [AZBA08], [NAA09]) that investigated general purpose Q&A websites "suggest that [they] may be poorly suited to provide high quality technical answers" [MMM+11], in practice, Q&A sites for programmers and software engineers are filling "archives with millions of entries that contribute to the body of knowledge in software development" [TBS11], where social media is playing an important role in developing code [STvDC10]. Frequently, on Q&A websites developers find suggestions, articles, discussions or even entire solutions for the problems they are dealing with and the huge amount of entries that technical Q&A provide constitute a base knowledge for example-programming practices. Well known solutions for common problems and ready-to-serve implementations provided by code examples help newcomers to have a better understanding of the programming issues and the context in which they are developing. Indeed, Q&A websites often become a substitute for official product documentation whenever it is sparse or it does not exist yet, being particularly effective at answering novices' questions and explaining conceptual issues [TBS11].

A prominent example of technical Q&A service is Stack Overflow[1]. This website has gained popularity among developers and it is becoming a usual means to help them in solving programming issues. Its growth since 2008 brought birth to the Stack Exchange network[2]. This network provides various Q&A services that aim not only to answer questions related to programming issues but also to maintenance purposes (*e.g.,* Server Fault[3]) and to software-unrelated topics (*e.g.,* Android Enthusiasts[4]).

The crowd knowledge provided by a Q&A service, such as Stack Overflow, can be leveraged to provide useful insight to developers. Even though the support given by the discussion in Q&A websites comes from outside the context of the project, it could be a valuable support as well: for example, programmers could tailor the information given by such external contribution and use it to fix issues they are dealing with or understand an external library in use. If we consider the case of Aptana[5], the project's discussions are totally hosted on Stack Overflow. In that case, both project developers and external programmers collaborate and debate issues and ideas.

However, the accessibility of such Q&A services is disconnected from the IDE. Developers can only use an external application, such as the web browser, to query and extract information. The disconnection of Q&A websites is one of the limitations that hinders their full potential and adoption in the software development process. The employment of RSSEs with this new source of knowledge is a viable solution to overcome the aforementioned limitation. They can bridge the gap between the IDE and Q&A services by providing suggestions that could be used and tailored both for code comprehension, programming issues, writing new parts of a system and debating design ideas.

[1] `http://stackoverflow.com`
[2] `http://stackexchange.com/`
[3] `http://serverfault.com/`
[4] `http://android.stackexchange.com/`
[5] `http://www.aptana.com/`

1.1 Contributions

We claim that the integration of those Q&A services in the IDE is a valuable solution to support developers in the software development process. Devising a recommendation system to provide suggestions for the programming issues could remove the current limitations of the interaction with Q&A services. In this thesis we present Seahawk [6], a plugin for the Eclipse IDE[7], whose main contributions are the following:

- Stack Overflow crowd knowledge integration in the IDE.
- Linkage between relevant discussions and source code through an ad-hoc annotations system.
- Sub-versioning system integration to provide developers with collaborative support, exploiting valuable Q&A in the context of team work.
- Example-based programming by allowing developers to retrieve code snippets from Stack Overflow Q&A directly in the code editor.
- Automated creation of queries from source code entities.

1.2 Structure of the Document

- In **Chapter 2** we discuss the related work concerning Q&A services and how crowd knowledge has been leveraged so far in the software development process.
- In **Chapter 3** we present our approach towards Q&A services, discussing how they can be harnessed to become part of the software development process.
- In **Chapter 4** we present Seahawk, a plugin for the Eclipse IDE that implements our approach by integrating Stack Overflow in the IDE.
- In **Chapter 5** we present the evaluation of Seahawk by discussing the results obtained in the experiments we devised.
- In **Chapter 6** we discuss the conclusion regarding our work and we present the future work.

[6] http://seahawk.inf.usi.ch
[7] http://eclipse.org

Chapter 2

Related Work

In this chapter we present the related work concerning this thesis. In Section 2.1 we discuss the studies on Q&A services and the role of social media in the software development process. In Section 2.2 we present the state of the art regarding the crowd knowledge integration in the development process and how it can be harnessed to provide a recommendation system.

2.1 Social Media & Software Development

Our work has been influenced by Q&A online services and how they can be leveraged to improve developer productivity. Section 2.1.1 shows studies on Q&A services and their role in the software development process. In Section 2.1.2 we discuss the features of Stack Overflow, a prominent example of Q&A service. Finally, in Section 2.1.3 we explain the limitations of Q&A services.

2.1.1 Studies on Q&A

Q&A online services have gained popularity in the last years not only among users but also among researchers. The software engineering research field is taking interest in Q&A services to understand how and to what extent they can contribute to the process of software development.

Treude *et al.* [TBS11] presented an exploration of the Stack Overflow service by analyzing randomly sampled data of the November 2010 data dump. They identified five categories of tags used in Stack Overflow by its community: programming language, framework, environment, domain, non-functional, and homework. The *homework* category is the only set comprising just one tag called *homework* as well. They defined a classification for questions in 11 different categories. They showed how categories such as *how-to*, *discrepancies* and *environment* were the most frequent ones, while *review*, *conceptual*, *how-to* and *novice* were the more answered categories. According to those findings, they claimed that Stack Overflow is particularly effective for code reviews, for conceptual questions, and for novices.

In their subsequent work, Treude *et al.* discussed the impact of web content curated by the crowd for software developers and their working practices [Tre12]. They raised questions about whether and how a large valid amount of programming knowledge could redefine the attributes of good programmers, about who owns the intellectual property of shared code, and about the impact of social media programming knowledge on software engineering education and career planning.

Storey *et al.* [STvDC10] also discussed how the use of the social media mechanism influences the software development practices. They posed and discussed questions with the aim of finding answers for the innovation of future software engineering tools. They tackled discussions about the role of social media usage in software engineering, how it can facilitate project coordination and improve individual development activities.

2.1.2 The case of Stack Overflow

Among the available technical Q&A websites, Stack Overflow[1] is becoming one of the most visible venues for sharing knowledge on software development [MMM+11]. By analyzing the data from the last public release (December 2011)[2] of the entire Stack Overflow data dump we measure approximately 880,000 registered users, about 2.3 million posed questions and 5 million answers, of which more than 1.4 million were accepted as resolutive from the person who posed the question. Mamykina *et al.* reported that more than 92% of the questions on expert topics are answered in a median time of 11 minutes [MMM+11]. Moreover, Treude *et al.* pointed out how "Stack Overflow is particularly effective for code reviews, for conceptual question and for novices" [TBS11].

Stack Overflow is designed by following nine specific design decisions [3]: voting, tags, editing, karma, pre-search, badges, Google is UI, critical mass and performance. The Stack Overflow service aims to differentiate its method to access knowledge from those offered by web search engines. This is mainly achieved through five of the aforementioned design decisions that guided the creation of the service: voting, tags, editing, karma and pre-search.

Voting addresses the critical issue that arises when reusing code samples found on the web: Before integrating a search result the developer has to assess its trustability to take a decision [GK10]. In the case of search engine results, developers have to take a decision only based on their knowledge and experience. In Stack Overflow, on the contrary, developers can also rely on the result of the voting mechanism: Site members can vote to approve or disprove any answer, and the sum of the votes creates an overall quality score based on crowd knowledge.

Tags explicitly state the technology and the topic to which the question and answer applies. When retrieving explanations and examples from generic websites, in fact, the referred technology might be unclear. In Stack Overflow, users are forced to attach at least one tag per questions allowing to retrieve and query the search engine through tags. Moreover, Stack Overflow provides a synonym system to cope with multiple tags with the same meaning (*e.g.,* lift and liftweb). Tags are the core feature of the query syntax in stackoverflow.com. Whenever a user performs a search, the inserted query is tokenized to identify possible tags and filter questions.

Editing clearly distinguishes this Q&A site from traditional web sites and web fora. Stack Overflow site members can freely edit questions and answers to refine them over time, thus creating a reliable and correct knowledge base that can be referenced and accessed later. The Stack Overflow community encourages editing contributions, allowing the person who asked the question to give also an answer, thus creating a sort of mini-blog entries [TBS11].

Karma encourages users' contributions. The user's karma value is determined by the number of rewards, called *badges*, users are assigned with. Every badge (*i.e.,* bronze, silver or gold badges) corresponds to a certain threshold of contribution reached: the more the user asks and replies to questions in a useful way (*e.g.,* the answer gets a high vote) the higher the reward is and the higher the privileges the user gets.

[1] `http://stackoverflow.com`
[2] `http://www.clearbits.net/torrents/1881-dec-2011`
[3] `http://www.youtube.com/watch?v=NWHfY_lvKIQ`

Finally, *pre-search* helps avoiding duplicates at a question's creation time. Whenever users have just finished to type the question, titles of similar entries are shown such that they can stop creating the question and solve their issue otherwise.

2.1.3 Limitation of Q&A services

Despite the aforementioned valuable features and its exceptional growth in usage and interest, we currently see two limitations in Q&A sites (*e.g.,* Stack Overflow) that hinder their full potential for software engineering.

1. Users should be able to focus only on the current task without any major interruption or disturbance [Ras00] but the interaction with Q&A online services is totally disconnected from the IDE (Integrated Development System). The interaction is separated from the context in which developers are performing programming tasks. LaToza *et al.* investigated developers' work habits [LVD06] pointing out how developers spend most of their time in the IDE, not only when writing code, but also when understanding it, and even when designing new parts of a system.

   Despite this, Q&A websites are currently only accessible from web browsers, disconnected from the development process. This might limit a consistent adoption of Q&A sites as a means to acquire knowledge: since Q&A sites are not integrated in IDEs, developers are forced to interrupt their flow and change context every time they need to deal with them. Moreover, there is no possibility to easily retrieve answers directly related to the current programming context, since the action of retrieving answers relies on the ability of the user to write the appropriate query.

2. Q&A websites provide a platform for questions aimed at "a general audience that is not part of the same project" [TBS11]. We do not see this as a limitation of the interest that a whole team working on the same project can have toward certain valuable discussions. In fact, we can imagine a developer who bases her implementation of a part of a software system on a meaningful discussion that took place on a Q&A site. Knowing the existence of such a productive discussion would be a valuable resource to recover the rationale, design, or implementation details of that particular portion of the system. Nevertheless, Q&A websites currently only offer web links to refer to their data and do not offer any resource to collaboratively and privately exploit valuable questions and answers in the context of a team working on the same project. This reduces the usage context of Q&A sites.

2.2 Crowd Knowledge & Recommendation Systems

Robillard *et al.* discussed Recommendation Systems for Software Engineering (RSSE), showing how they provide support to developers by tailoring data extracted from analysis, and discussing the general architecture of a typical RSSE. As they state, "An RSSE is a software engineering application that provides information items estimated to be valuable for a software engineering task in a given context" [RWZ10]. According to this definition, our work falls in the context of RSSEs.

We consider the crowd knowledge a valuable resource to harness and to integrate in the IDE, to help developers in the software development process. The integration of crowd knowledge in the IDE has already been tackled by research in the past. Initial steps towards the integration of in-project knowledge were made by Laugher and Rodden [LR93] by developing an annotation system.

This system allowed to link documentation and discussions regarding design decisions at source code level. Developers can access this knowledge through ad-hoc views that provide additional information regarding the meaning of a code entity. Whenever a code entity is selected, the information is shown in different forms (*e.g.,* additional description, graphs *etc.*) and helps the developer who has no knowledge of the system (*e.g.,* newcomers in a project). Even though this approach is good in assisting developers in understanding the code, forcing them to look at the code to get additional information is a drawback. The information should be accessible even if the developer is not working on the current entity.

Bacchelli *et al.* presented ReMail [BLH11], a plugin to integrate crowd knowledge from mailing lists in the Eclipse IDE. In their previous work [BLR10], they employed information retrieval techniques to extract information from emails and to consequently link them to source code entities in a lightweight fashion. Given a code entity (*e.g.,* class), the ReMail's recommender system suggests emails to developers such that the knowledge they contain is accessible from the IDE.

Holmes *et al.* presented DeepIntellisence [HB08], a plugin for the Visual Studio IDE [4] that links bug reports, emails, events history (*e.g.,* checkins, code changes *etc.*) and people to source code entities. DeepIntellisense integrates all this information in three different views that are updated accordingly to the entity under the cursor in the code editor.

Čubranič *et al.* investigated the usefulness of project's group memory for newcomers and created Hipikat, an Eclipse plugin that "provides developers with efficient and effective access to the group memory for a software development project that is implicitly formed by all of the artifacts produced during the development" [ČMSB04]. Hipikat provides a recommender system to assist developers who are new to the project by recommending items from problem reports, newsgroup, articles *etc.*

The aforementioned examples focus on integrating crowd knowledge for a specific project in the IDE. Q&A services provide crowd knowledge that can be integrated in the IDE as well. The information retrieved among discussions provided by an external community can be used to solve issues. Even in the open source reality this practice is being adopted: Aptana[5] is an example in which crowd sourcing is used to provide support for developers, relying on Stack Overflow as a means for hosting discussions regarding their project. In this direction, other examples of tools and systems integrate external crowd knowledge in the IDE. They leverage the interaction of the developers to find out the task context and recommend relevant information.

Brandt *et al.* presented Blueprint, a plugin built on top of Adobe Flex Builder that integrates a web search interface. Blueprint "automatically augments queries with code context, presents a code-centric view of search results, embeds the search experience into the editor, and retains a link between copied code and its source" [BDWK10]. Blueprint is tightly coupled to the languages that Adobe Flex Builder is designed for (*i.e.,* Javascript and MXML); developers can import code examples just for those. Blueprint allows keeping up to date the imported code samples by means of an annotation system that works at the source code level. However, the link between copied samples and their source is language dependent as well. For this reason, Blueprint limits the scope of its action. To overcome such a limitation, Blueprint should provide developers with features to import code snippets and to annotate them in a language independent fashion.

Sawadsky *et al.* presented FishTail [SM11], a plugin for the IDE Eclipse that is built on top of Mylyn[6] (previously introduced as Mylar [KM06]). The goal of FishTail is to automatically suggest code examples from the web that are relevant to what the programmer is trying to accomplish.

[4] http://www.microsoft.com/visualstudio/en-us
[5] http://www.aptana.com/
[6] http://www.eclipse.org/mylyn/

To suggest code examples, they generate keywords from the program element name (e.g class, method or fields) that has changed the most, according to the MYLAR's DOI (Degree Of Interest) policy. They rely on Google as a means to retrieve documents, creating four different queries for four different targets (*i.e.,* article, blog, example and tutorial). Generating queries just from the name of the most used program element is a limitation. As they suggest in their work, this approach seems to point out relevant documents just in case of interfaces. One possible improvement to that approach is to enrich the query with topics extracted from one or more program entities, providing users with the capability of deciding which entities are interesting for their issues. Eventually, this approach could also reveal some hidden relationships among entities that cannot be spotted from code analysis.

Finally, Goldman *et al.* presented CODETRAIL, a system that connects source code with web resources by exploiting a communication channel and a shared data model to implement a variety of interactive tools [GM09]. The communication relies on an ad-hoc backbone where a plugin for Eclipse IDE is present at one end and a plugin for Mozilla Firefox [7] is present at the other. CODETRAIL harnesses information available in the IDE and in the web browser (*e.g.,* editing history, code contents and browsing history) to tackle the problem of the context switch by eliminating the effort required to synchronize the two applications. Therefore, the browser, or the IDE, is focused on the relevant source code, or web page. Even though the web browser provides additional functionalities, relying on an external application limits the user interactions. For example, code snippets can only be imported manually. On the contrary, a full integration in the IDE can help developers to manipulate information contained in the web pages and use it for their purposes.

The related work defines the initial context from which our work takes place. All the aforementioned limitations can be tackled by implementing the discussed improvements. In the next chapter we illustrate the approach we followed to enhance the interaction with Q&A services and the additional features developers should be provided with.

[7] `http://www.mozilla.org/`

Chapter 3

Approaching Crowd Knowledge

In Section 2.1 we discussed the limitations of Q&A services. We pointed out how those services suffer of disconnection from the IDE, thus lowering the developers' performance. In this chapter we present the concepts behind our approach: we discuss what we consider a valuable process to cope with those limitations and offering a better interaction with Q&A services to the developers.

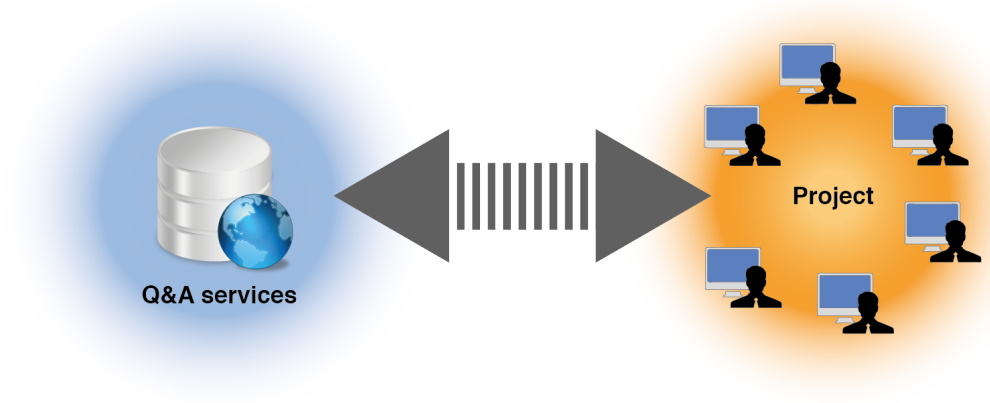


Figure 3.1. The conceptual representation of our approach

As shown in Figure 3.1 project development and Q&A services are the two key actors. Q&A websites and project development can be considered as two separate worlds. If developers need to access online resources, there is no direct connection with Q&A services. On the contrary, they have to travel between the two spaces, search for the needed information, and come back to the development phase. If we think about the current state of the art, as already discussed in Chapter 2, the absence of a direct integration in the IDE of such a source of information (Q&A) causes the context-switch between the two spaces. Our approach is the link between the two, represented by the grey arrow in Figure 3.1. Instead of letting developers mine the Q&A entries, we should provide a means that leverages and tailors such information to make it available in the IDE.

If we think about Q&A services, we can imagine them as a huge database built of millions of entries in which a multitude of topics are discussed. Among them, developers can crowd source solutions and even tasks that can help dealing with issues during the software development process.

Initially, we must eliminate the web browser as a vehicle to get those services. The developers should be provided with the capability of querying such a database from the IDE itself. It would be useful for a developer to open up a new view in the IDE, write the query and get the related documents. As a second step, we have to enhance the interaction between the content of the document and the developer. In technical Q&A services, the code can be tailored to get a partial implementation which compiles and works. Without any support tool, developers need to locate the code, copy and paste it in the editor before starting to modify it. Providing them with an ad-hoc view that automatically locates code snippets and allows to easily drag and drop them in the editor could be a valuable enhancement. Finally, the IDE itself should be able to provide support in building the query by understanding the context in which the developer is writing code. For example, the user should be able to automatically generate a query from the source code, adding additional information as needed, and retrieve entries from the Q&A services.

In a project, developers do not only work alone but they cooperate as well. Similarly, in Q&A services users collaborate, even without having a common end. Another valuable feature a developer should be provided with is the possibility to share the crowd knowledge with teammates. At the moment, developers can just share this information via communication means such as email, instant messaging *etc.* On the contrary, this kind of knowledge sharing should be provided directly in the IDE: A recommendation system could suggest the documents already added by teammates and notify new entries to the developer without having to leave the IDE.

The aforementioned enhancements are the base of our approach. In the next chapter we give a detailed presentation of how those ideas take place in reality. We propose the approach we discussed so far by implementing a tool to support developers: Seahawk.

Chapter 4

SEAHAWK

In this chapter we present SEAHAWK and its backbone system. We give an overview of the SEAHAWK's architecture with a detailed description of the plugin side's components, we discuss the data collection mechanism of the SEAHAWK backbone, and finally, we present the User Interface (UI) elements of the Eclipse plugin we built.

4.1 The Architecture

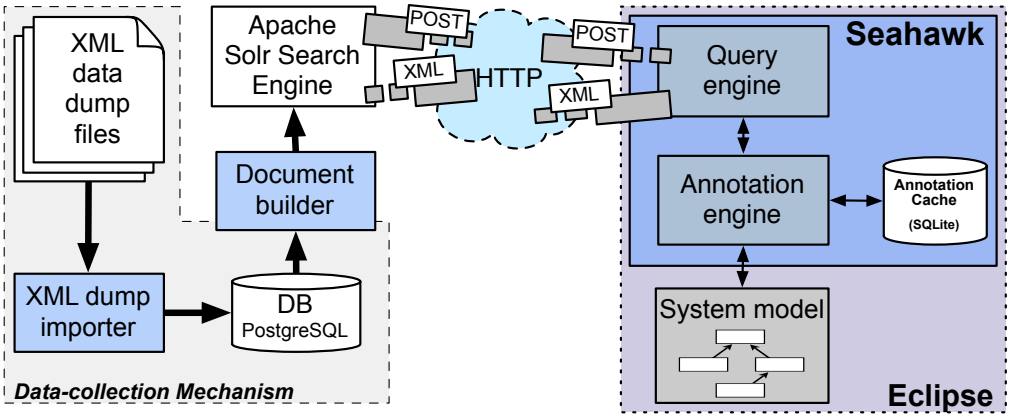


Figure 4.1. The SEAHAWK architecture

According to Robillard *et al.*, a recommendation system is generally composed of three main components: *A data-collection mechanism*, a *recommendation engine* and a *user interface* [RWZ10]. We follow this schema to present SEAHAWK's architecture depicted in Figure 4.1.

As we explained in the previous chapter, we want to build a recommendation system that provides suggestions from Q&A services' crowd knowledge. The first step towards this direction concerns the definition of a data collection mechanism. This component in the SEAHAWKsystem is the one responsible of gathering Q&A from Stack Overflow. As we discuss in Section 4.2, the Stack Overflow data is retrieved through a set of XML files.

The data contained in the files is then extracted through the *XML dump importer* and put in a relational database. We built a tool to query such a database and reconstruct a JSON representation of each document in order to make it available for any language. This representation is then included in an additional document schema required by the Apache Solr[1] search engine. When documents are indexed by the search engine, they become immediately available for any query. Apache Solr provides a RESTful interface to perform searches by mean of both GET and POST requests and it replies with XML data containing the relevant documents. For further details regarding the data-collection mechanism, see Section 4.2.

On the other side of the Seahawk system chain we implemented an Eclipse plugin. Its main purpose is to implement the recommendation engine and the user interface. The interaction with a recommendation system can be both manual (*i.e.,* a query is inserted by the user) and automatic (*i.e.,* the recommendation engine generates the query) [RWZ10]. In Seahawk we implemented both: the user is provided with an ad-hoc view to manually insert queries (see Section 4.4.1) as well as queries that can be provided by Seahawk itself by analyzing the code (see Section 4.3.1). Seahawk also provides an annotation system (see Section 4.3.3) that allows developers to annotate Stack Overflow documents directly in the code. Developers receive recommendations for the documents annotated in the code via an ad-hoc view (see Section 4.4.2) and they get notified whenever another developer has put a new annotation in the code (see Section 4.4.4).

4.2 Data-Collection Mechanism

This section presents the work behind the data collection mechanism we implemented in the Seahawkbackbone. We initially present the possible ways to gather data and our adopted solution. We provide an analysis of our document modeling idea, how it is designed to support more than the Stack Exchange documents, and the data manipulation process, taking also a look at the advantages and drawbacks of the solution adopted.

4.2.1 Stack Overflow Data Pre-processing

The first step to accomplish in building the Seahawk backbone concerns the data collections mechanism. Even though getting data from a website or a web service seems to be an easy task, this does not apply for the Stack Exchange network, in particular for Stack Overflow. As explained in Chapter 1, the size of the Stack Overflow data dump is not negligible. There are many challenges to take into consideration when dealing with such a huge amount of data. Moreover, the level of difficulty increases when the data needs to be manipulated or a new representation of the information is needed. We evaluated different solutions to collect data and perform searches among the millions of documents stored in the Stack Exchange network.

In the first place, the Stack Exchange network provides a public RESTful API[2] to interact with its own data. The network does not only comprise of Stack Overflow, but it consists also of a series of websites that can be taken into consideration for the goal of this thesis (*e.g.,* gamedev.stackexchange.com), as well as other websites treating completely unrelated topics (*e.g.,* cooking.stackexchange.com and android.stackexchange.com).

[1] `http://lucene.apache.org/solr/`
[2] `https://api.stackexchange.com`

The data of all of web sites are accessible through the API. such that every information regarding users, posts, tags *etc.* can be retrieved in JSON format. Such a web service can be useful to facilitate the data collection but not to search for a document. As the API states, the searching capabilities are limited on purpose to just retrieve documents using, for example, title matching. They also suggest to implement a custom search engine or take advantage of another available one which focuses on just the Stack Exchange website of interest.

However, the web service could be still leveraged as a data source to reconstruct a local, periodically updated, database. This solution would not require a difficult manipulation of data or a high degree of maintenance: Due to the JSON format used by the service, it would require just a crawler to retrieve documents and a non-relational database (*e.g.,* CouchDB or MongoDB) to store JSON documents. Unfortunately, the Stack Exchange policy regarding the usage is very limited. They allow one hundred queries per day to be performed from the same IP address.

In the second place, we could just discard the idea of creating a local database and decide to perform searches directly on the Stack Exchange website of interest. To retrieve the documents, it would require to implement a webpage scraper to parse the search results and extract the URLs of the questions. This solution requires little effort to be implemented as well as no local database and the related maintenance. However, this would limit our interaction with the raw data and it would give less additional information (*e.g.,* user information). Moreover, it sounds as an unstable solution since the maintenance of such a backbone would require a new Stack Exchange crawler whenever the page layout is changed.

As a last approach, Stack Exchange provides a public data dump of their data[3]. All of the websites' data in the Stack Exchange network are available and released every three months. The dump comprises several XML files that represent the dump of the database of the website. We can limit the files needed to the ones representing the data (*i.e., posts.xml, users.xml, comments.xml*), thus discarding the files regarding the evolution of the website (*e.g., posthistory.xml, badges.xml, votes.xml etc.*). Even though this could limit the amount of information available to study, we are more interested in the documents provided by Stack Exchange than the users' data regarding their interaction with the community. Extracting more information about users to define an accurate policy based on user's reputation and interaction in the community could be a valuable feature. However, this is out of the scope of this thesis and we can consider that as a future work.

Collecting data from Stack Exchange allows us to have full control on the data collected, thus deciding to take into consideration just what we consider useful. Nevertheless, dealing with XML files that are released every three months has some drawbacks: first of all, this approach can stand as long as the data dump is provided. If the policy regarding the distribution of such data changes, this methodology would not work anymore. Secondly, the process of importing and manipulating a considerable amount of data from websites such as Stack Overflow requires a significant amount of resources. As we discuss in Section 4.2.2, the whole process requires a relational database, a search engine and about 8 hours of data pre-processing. Finally, data is not up to date since posts are at least three months old. Although well known languages (*e.g.,* C++, C#, Java *etc.*) could be discussed enough to help with programming issues, newborn languages and technologies (*e.g.,* Scala, Haskell, Dart *etc.*) could suffer because of this update limitation.

All these drawbacks sound like substantial limitations. If, and only if, the Stack Exchange network would have provided an open and unlimited API, we could have avoided the pre-processing effort. However, this approach is the only way to have a real search engine accessing the data without limitation in usage.

[3] `http://www.clearbits.net/creators/146-stack-exchange-data-dump`

4.2.2 Data Import

Among the Stack Exchange websites, Stack Overflow is the one requiring more effort to manipulate data. According to the information extracted from the last public dump (December 2011) the total approximate size is about 9 GB. The XML file containing Q&A is the biggest one (~ 7GB).

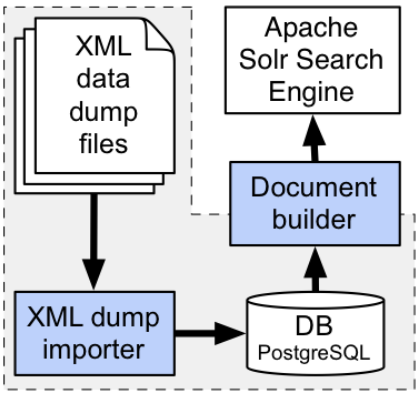


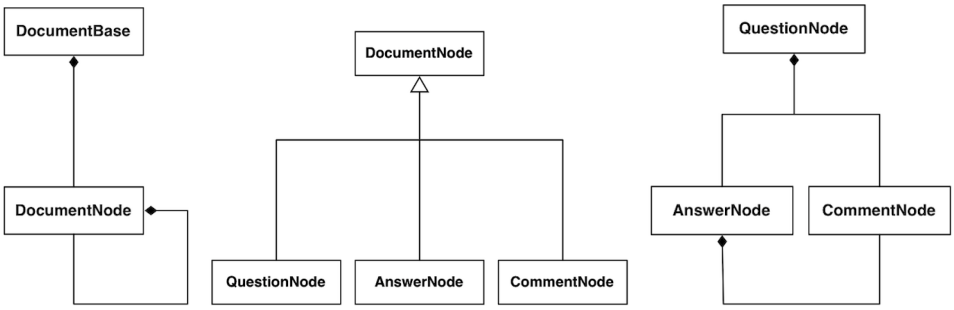
Figure 4.2. Data Collection Mechanism Process

In Figure 4.2 we see the process we devised to import and manipulate data to reconstruct documents that are then indexed by the search engine. We just consider three XML files: *posts.xml, users.xml, comments.xml*. Being a plain representation of the tables in Stack Overflow's database model, one of the challenges in extracting the documents is to join the data among those files. The total amount of entries in *posts.xml*, according to the December 2011 dump, sums up to more than 7 millions. To recreate a document, we need to gather a question and all the related answers. For each answer, but also for the opening question, we need to extract the users, the comments and their authors. Performing this operation by manipulating data directly from the XML files would require too much resources and time. This operation should be done by a relational database. For this reason, we decided to perform an intermediate import phase in which we partially reconstruct the Stack Overflow's database.

Once the database has been populated with the data contained in the XML files, the documents can be extracted. We chose to represent the document in JSON format in order to make them portable as much as possible. To perform this task, we implemented a importer that queries the database, thus extracting all the information needed to get complete documents. Those documents are then included in an additional document representation required by Apache Solr, whose details are discussed in Section 4.2.4. Finally, the search engine indexes the documents extracted by our tool, making them available for any research.

4.2.3 Document Model

The good feature of Stack Exchange network is that these websites have the same structure. To be tied to just one kind of service, we wanted to achieve a flexible model that allowed us to represent any kind of document. We have decided to represent this model in the JSON format. Storing and indexing the document in this format allows to deserialize to any language the JSON describing a document. Even if the target language in this thesis is Java, those JSON objects can easily be deserialized to C#. Thus the Seahawk's backbone could be also ported to Visual Studio.



(a) The document model (b) Example of document node extension for Q&A (c) Q&A document representation

Figure 4.3. SEAHAWK's document model examples

In Figure 4.3a we depict the model we use to describe a document. From a conceptual point of view, a document can be seen as composition of document nodes. This way of representing documents is commonly used in XML languages. Extending that idea to textual documents is not hard. For example, a book is composed of chapter nodes that can have section nodes as children. The same applies for Q&A documents (see Figure 4.3c). Take into consideration the case of Stack Overflow: A question defines the beginning of a document and the answers can be seen as children of the opening question. Comments can be considered children as well for both question and answer nodes. Every additional information can be added by specializing the *DocumentNode* class as in Figure 4.3b. The *DocumentBase* class represents the general information regarding the document (*e.g.,* title, author *etc.*), and it contains the root of the document's tree. Thus, in case of Q&A documents, an instance of *DocumentBase* contains a *QuestionNode* as a root.

One of the good aspects of such a model is that every document represented become traversable as well as an XML. This feature is useful to build a navigator for those documents, thus allowing users to jump from one point to another. To that aim, in Section 4.4 we present how one of SEAHAWK's view can take advantage of this representation to provide a useful interface to traverse the document.

4.2.4 Search Engine

The main purpose of the search engine is to index documents whenever a document is extracted and reconstructed from the database, and to make them available for queries. Rebuilding from scratch a search engine would have required a huge effort. In particular, the reliability of the search system should have been tested extensively to guarantee its effectiveness.

We chose to take advantage of Apache Solr instead of developing a search engine from scratch. Apache Solr stores and indexes documents in a vector space model, relying on Apache Lucene [4] as core engine. The weighting algorithm used by Apache Lucene, and thus by Apache Solr, is a variation of the standard *tf-idf* [Hat10]. To use such an implementation, we needed to define one more document schema. Apache Solr requires an XML schema that describes the fields composing a document, what must be indexed, and which one is the unique identifier for a document. Moreover, the schema defines what kind of text processing must be performed on those fields at indexing time and at query time.

[4] `http://lucene.apache.org/`

In our case, we defined the subsequent fields in the schema:

- **Id**: The unique identifier of the document.
- **Title**: The title of the document, that is, the question for a Stack Overflow document.
- **Document**: The JSON representation of the document.
- **Tag**: The list of tags, if any, describing the document.

According to that schema, we replicate data on purpose. Fields such as *Title* and *Tag* are already available in the JSON representation contained in the *Document* field. However, this replication is needed to enhance the search functionalities. As we explained in Section 4.1, the SEAHAWK's query engine does not search only in the *Document* field. The actual configuration treats *Id*, *Title* and *Tag* as strings, while *Document* is considered text. Thus, information retrieval pre-processing is applied to just the *Document* field. We configured Apache Solr to remove stop words, to filter out possessive words, to stem words, to trim white spaces, to filter synonyms and to lower the case (see [MRS08]).

We tried to apply tag synonym filtering as in Stack Overflow through the synonym filter of Apache Solr. The list of synonyms can be retrieved by the API provided by Stack Exchange. To improve the indexing phase and to enhance the tags contained in the document's text, we used the same tag synonyms list as Stack Overflow does. Tags often define the technology discussed in the posts (*e.g.,* java-se, .net *etc.*), thus enhancing them is worth being done. We implemented a crawler to retrieve the tag synonyms list from the Stack Exchange API. We noticed that for some technologies that requires a version number, the list generalizes the number by putting an "x" (*e.g.,* python-3.x). Since document tokens have to match the tag text, and considering that it is unlikely to have users writing the version number in that way, we applied the following approach: we removed the ".x" or the "x" from the tag string and we enlarged the synonyms set to match the versions ranging between 1 and 9. Thus, synonyms like "python-3.1", "python-3.2" *etc.* are indexed as "python-3". This applies also at query time: If a user searches for a specific topic and puts a tag synonym in the query, for example "python-3.5", the synonym is considered as "python-3".

An advantage of having the Apache Solr schema resides also in the field *Id*. For that field, we decided to use the original URL. This allows to put together Q&A taken from every website in Stack Exchange. Having a unique id to distinguish documents allows to perform online updates without interrupting the web service provided by the search engine. If Apache Solr indexes a document that has been already indexed, it just overwrites it. Once the indexing phase is complete, the Apache Solr engine can be queried via HTTP in a RESTful fashion. The SEAHAWK plugin can thus query the search engine to get relevant documents in XML format and to extract the JSON object from the *Document* field, deserialize and show its content.

4.3 The Recommendation Engine

The recommendation engine of SEAHAWK provides both manual and automatic interactions. The core is divided in two main engines that provide developers with the possibility of querying the crowd knowledge external to the project (*e.g.,* Stack Overflow) or to suggests documents to other developers working in the same project. In this section we present the query engine and the annotation engine.

4.3.1 Query Engine

SEAHAWK's Eclipse plugin makes the Q&A crowd knowledge available in the IDE. The users can interact with this knowledge in multiple ways that the website normally does not allow to. We explained in Section 4.1 that the data-collection mechanism exposes a search engine (Apache Solr) as a means to retrieve the Q&A services data. The main goal of the query engine is to communicate with Apache Solr, thus creating a query given an input string. As presented in Section 4.2, Apache Solr uses a document schema that defines fields to be used in a query. The document schema we defined comprises four fields: id, title, document, tag.

The query engine focuses on just the *document* field and the *title* to build query for Apache Solr. Being Q&A documents the target of such queries, it is likely to have some information also in the title. It is worth giving a different weight to the document's title in order to exploit possible keywords that can be relevant for the target search. We now present how a query is built by mean of example. Assume that a developer wants to query the search engine with the following query: "change label color in Java". The query engine takes the string inserted by the developer and tokenizes it. Then, the engine builds the query, according to Apache Solr syntax, in a way that every token must be present in the document field or at least one of those is contained in the title field. Thus, the resulting query is the following:

(document:change AND document:label AND document:color AND document:in AND document:Java) OR (title:change OR document:label OR title:color OR title:in OR title:Java)

In this query the overall relevance of a document is determined by the relevance of the body of the document and its title. Documents whose title is interesting for the query researched even if the document's content does not match any of the tokens are enhanced anyway. For a simple query as the one used in the previous example, the title can be a key factor to retrieve a document that can solve a programming issue. Even on Stack Overflow, the first result given for such a query is "How can I change label color in Java?" that is exactly what a developer could have looked for. In those cases, the document body could be discarded. From a certain point of view, this approach can lead to poor results. For a particular query that has no relevant document to be associated with, documents could still be found. However, the documents that matches only the title would have a low relevance value given by Apache Solr. To create the query we avoided text processing on the tokens on purpose. Stop words such as "in", are left in the query because we rely on a post-processing phase made by Apache Solr on the server side. The only pre-processing made on the query inserted by the developer regards the reserved words of the query syntax. For example, words like "AND", "OR" and symbols can easily break the query, thus leading to a failure. To avoid these problems, we filter those words from the input string.

4.3.2 Automation of Queries

The part of the query engine explained in Section 4.3.1 is responsible for the manual interaction with the recommendation engine. None of the previous features explained allows to automatically generate a query but it assumes that someone has built the query (*e.g.,* developers). In SEAHAWK we want to provide developers with the support of an automatic tool to extract relevant keywords from the code to build a query.

In Figure 4.4 we depict the internal structure of the query engine. Section 4.3.1 discussed the functionalities of the component at the top of structure (*i.e.,* the *Solr Query Syntax Adapter*). We now focus on the rest of the structure, presenting the mechanism behind the automatic keywords extraction. The first technical issue to overcome regards the code written by developers.

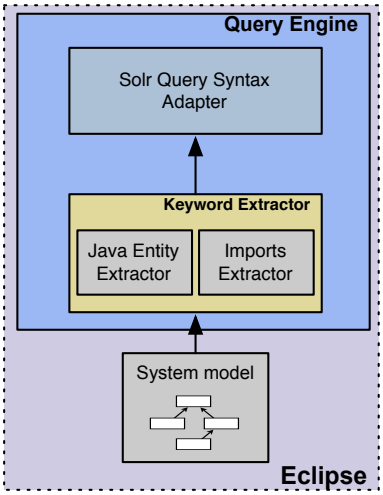


Figure 4.4. The query engine components

It is likely that developers need to understand their code even though it does not compile at all. Dealing with code that does not compile brings some limitations. If compiling code can provide a full Abstract Syntax Tree (AST), in case of compilation errors the AST can be partial or even absent. Moreover, the partial AST is the representation of the code until the compilation failed and it discards any possible compilable code that comes after. That applies also the Eclipse IDE when its framework is asked to produce an AST for a Java program. Without the chance of having an AST to extract structural information from the code, a way to get at least some coarse grained information is to perform island parsing. Bacchelli *et al.* implemented a similar strategy to extract Java code snippets from emails [BCLM11]. They were able to extract class definitions, method definitions, method invocations, stack traces *etc.* from text with natural language. Having source code is not that different: the uncompilable code plays the same role as the natural language and it can be ignored by the parser. We can thus use island parsing techniques to extract structural information to leverage. Fortunately, the Eclipse IDE already provides a framework to apply similar parsing approaches to Java code: It allows to identify classes, methods and fields in a source file even though the compilation fails. For the approach followed in this thesis, we do not consider fields but we just extract keywords from method and classes.

Differently from all other features of Seahawk, the automatic query generation is the only feature to be Java dependent. Seahawk provides developers with the possibility of selecting a group of Java entities (see Section 4.4.5) to extract keywords from. Among the entities, developers can also select packages or compilation units, but only the classes contained in those are considered targets for keywords extraction. Seahawk also allows to select an entity from the cursor position in the editor. Since we are able to identify the entities in a source file, we can thus detect their position in the text and evaluate which one is the target entity by checking the cursor's offset.

When an entity is selected, the query is built by merging the keywords obtained by two kind of extractions: import analysis and entity's body processing. The former extraction concerns the creation of a set of keywords from the import statements. To accomplish that, we take all the import statements in the source file and we filter out the ones that are not used by the target entity.

We identify the used imports by applying a naïve matching on the class name: If the class name is contained in the entity's body, we consider this import or we discard it otherwise. This approach can lead to false positives in case two classes have the same name, but reside in different packages, are used by the same entity. However, we believe that such situations rarely happen. Once identified the imports, we proceed by tokenizing each statement on the "." character, and by building a set of unique tokens that become part of the query. For example, assuming we have the following import statements used by an entity:

```
import java.util.List;
import java.util.ArrayList;
```

The resulting set of tokens is [*java, util, List, ArrayList*]. The latter extraction concerns the entity's body. We apply some information retrieval techniques to extract the ten most frequent keywords in the body. First of all, we tokenize the entity's body on white spaces. For every token obtained, we split it on case change, digits and symbols. Finally, we lower the case and remove stop words. The set of tokens we obtain is then ordered by frequency and the first ten are taken into account to become part of the query. To this set of keywords we always add the entity's name. The reason why we also take into consideration the name of the entity is due to the Java interfaces. If the entity is a method, including the name would enhance the research. Being fixed, the method's name of a Java interface in a library, or framework, is always the same. A Stack Overflow document would use this method's name if one of the code snippets is tackling the implementation of a specific interface.

We believe that the combination of information extracted from imports and entity's body could provide a good overview of the topics and the context of the development. We focused on imports because, in a Java application, they somehow define the context in which the developer is working. For example, from the imports we can understand which libraries are used or what framework the developers is programming with. This information is valuable to filter out discussions in Stack Overflow. Looking for the class names used in a library, or framework, could lead to specific discussions or code snippets. Moreover, we also believe that topics taken from the entity refines the scope of the research. For that reason, we use all the tokens from the import statements to then add the keywords of the target entity.

4.3.3 Annotation Engine

Apart from integrating crowd knowledge from Stack Overflow in the IDE, we want to make developer able to collaborate by means of the crowd knowledge itself. To this aim, we implemented an annotation engine to provide developers with the possibility of putting annotations in the code. This system is one of the core part of Seahawk's recommendation engine. Differently from the query engine, that provides automated query generation, the annotations engine implements the second aspect of the manual interaction in the recommendation system. It exploits the collaboration among teammates to filter out and suggest documents from Stack Overflow's crowd knowledge.

In Figure 4.5 we depict the main components of the annotation engine. There are two main purposes in the annotation engine: creating and parsing annotations. In order to be language independent, the annotation structure must be flexible. We wanted to achieve this flexibility by embedding annotations in multi-line comments. A similar approach has been followed in Doxygen[5] to integrate documentation in, for example, C++ and Java code and in Blueprint [BDWK10] to link code examples to code. Both of them enclose meta-information between multi-line comment delimiters (*i.e.,*/* and */) and define fields by putting "@" as prefix character.

[5] http://www.doxygen.org/

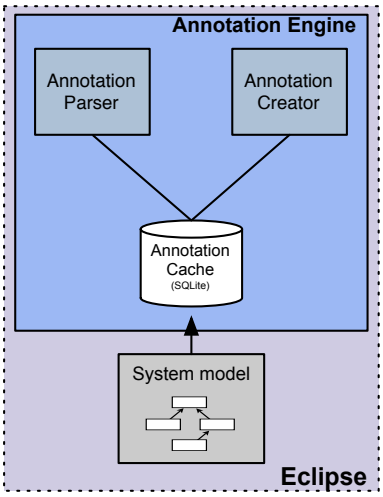


Figure 4.5. The annotation engine components

The Seahawk's approach follows the same guideline but it enlarges the scope by including more than languages whose multi-line comments uses, for example, Java's delimiters. To that aim, we allow developers to define custom delimiters that, of course, need to match the target language' syntax for comments. For example, Python multi-line comments can be enclosed in delimiters for strings (*i.e.,* """ and """), and the same applies also for XML (*i.e.,* <!– and –>). A developer can define custom delimiters for Seahawk by just extending those delimiters with additional characters.

To not conflict with Doxygen or JavaDoc annotations, we decided to put an exclamation mark as last character for the opening delimiter. For example, in Java the opening delimiter would become /*! (instead of /*) while in XML it would become <!–! (instead of <!–).

Listing 4.1. Example of Seahawk's annotation for Java/C++ languages

```
/*!
* @documentId http://gamedev.stackexchange.com/questions/1901
* @title Unit Testing a C#/XNA Game Project
* @comment example comment
* @author Seahawk
* @creationTime 2012.04.13 17:29:13.460 CEST
*/
```

Listing 4.1 presents an example of Seahawk's annotation. The annotation is very similar to Doxygen's. The meta-information represented is limited to the necessary: whenever an annotation is created, we report the id of the document, its title, a comment put by the developer, the author of the annotation and the creation time. The id of the document allows to identify the target document to be suggested (see Section 4.4.2). The other fields are used to implement the basis of the collaborative part. Seahawk does not use any backbone to provide collaborative functionalities. It just relies on the fact that a versioning system (*e.g.,* Git, SVN *etc.*) is used in the development phase. Putting annotations in the code in enough to keep track of the document suggested by developers, thus linking documents to a specific revision of the source code.

The whole collaborative process is embedded in the normal development phase: whenever a developer commits, the annotations are committed too.

The role of the *comment*, *author* and *creationTime* fields is to guarantee that annotations are unique. The *comment* field is mainly used to allow developer to communicate with each other through the annotation system. Whenever a developer updates the repository, the new annotations are updated together with the comment explaining the purpose of the linked document. We want to be able to distinguish annotations among themselves and to have the same document linked in different part of the same source code file. For that reason, we introduced the *creationTime* field: It allows to distinguish an annotation from another even though they have the same value in the remaining fields. The granularity of the timestamp is maintained at milliseconds level on purpose. In doing so, this way of distinguishing annotations fails in just one rare case: Two developers must link the same document, with the same comment, having the same *author* value in the same millisecond. This is unlikely to happen.

To definitely exploit the collaborative functionalities, the annotation engine provides also a notification system to keep track of the annotations already seen by developers. For that reason we had to use two different ways of parsing code. We implemented our own parser for annotations as well as taking advantage of the partitioning system provided by the Eclipse IDE. The latter requires to declare delimiters for the interested partitions in the code. Thus, putting a third character allows to define an ad-hoc delimiter, as explained before, meeting the requirements. Whenever a source file is opened or modified in the code editor, the partitioning system notifies the view showing the suggested documents (see Section 4.4.2) and stores the annotations in the cache. The latter relies on our implementation of the parser that works in background whenever a project is updated, thus extracting annotations from the files being updated. The annotations obtained are then checked in the cache. If some of them are not present in the cache, they are notified to the developer (see Section 4.4.4).

4.4 User Interface Elements

In this chapter we present the user interface of Seahawk. We discuss the functionalities implemented by all views, how the User Interface (UI) takes advantage of the recommendation engine described in Section 4.3 to suggest documents inherent to the development context, and how they provide developers with collaborative functionalities.

4.4.1 Document Navigator View

The first view to discuss concerns the manual interaction with the recommendation engine. As depicted in Figure 4.6, in the document view the use can manually enter a query to retrieve documents from the Stack Overflow crowd knowledge. When interacting with any of the Stack Exchange websites, the results of a search is shown as a list of entries. There is no way to get any other information unless the users clicks on the entry and gets the Q&A document. We have already followed this approach in a preliminary version of Seahawk [BPL12]. We believe this approach is too limited, in particular if users have to check a huge amount of documents before getting the desired solution, or hints, for their programming issues.

For that reason, we decided to provide developers with a tree view. In doing do, they can easily navigate through the document without having to redo the query and jump from one part of a document to another part in another document. Initially, results are presented as a list of documents.

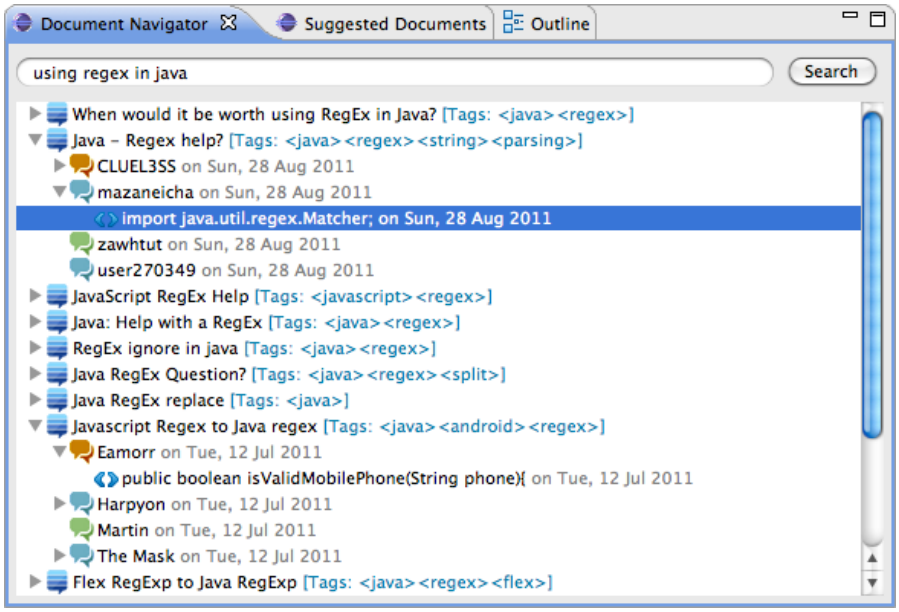


Figure 4.6. The document navigator view.

Each document shows the title and the list of tags on its right side while each node reports the author's name as title and the date in which it was created. Every time the user navigate in the tree and moves on a node, the content of that node is presented in the document's content view (see Section 4.4.3). Thus, a developer can visualize the entire document or a single node (*e.g.,* question, answer *etc.*) without having to read everything or scroll the document from the beginning.

The tree view used maps the model we discussed in Section 4.2.3: Every root node is a document (*i.e.,* DocumentBase instance) and its children are either questions and answers (*i.e.,* DocumentNode instances). As we can observe, we used colors to identify the different types of nodes. We assigned orange to question nodes, blue to answer nodes and green for the accepted answer. Question and answer node can have children as well. We avoid to put comments in the tree as children of them, but instead we show code snippets. The reason why we made this choice is related to the interaction this view provides developer with. As we explained in Chapter 3, we want to enhance example-programming practices by improving the way developers import code snippet from Stack Overflow. Moreover, we want to provide a means to exploit collaboration between developers through an annotation system (see Section 4.3.3). We believe that this view can be used to both purposes and we employed a *drag and drop* (D&D) interaction to achieve it. A developer can drag a document or a code snippet into the code editor. Whenever a document is dropped in the editor, Seahawk asks the user to put a comment via a dialog. The user can still decide to confirm the creation of the annotation or to rollback the operation. Whenever a code snippet is dropped in the editor, the code is automatically copied in it. The user can then start manipulating and tailoring the code to achieve the desired result. The other document nodes presented in the tree view cannot be dragged. We do not see the need of linking a specific node to the code, or to let developers import Q&A text as comment.

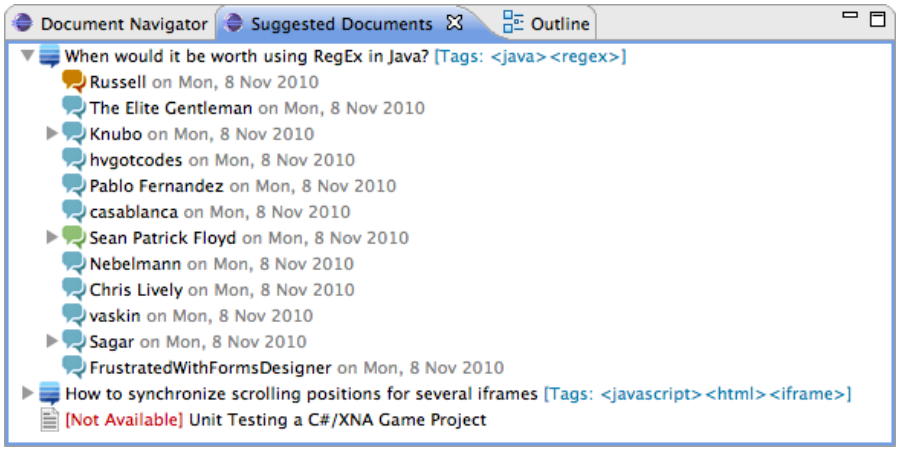


Figure 4.7. The suggested documents view.

4.4.2 Suggested Documents View

As we discussed in Section 4.3.3, Seahawk provides developer with an annotation system to link documents to the source code. As depicted in Figure 4.7 a tree view similar to the one presented in Section 4.4.1 is used. Instead of presenting documents retrieved from a query, this view tightly works with the annotation engine. Whenever an editor is brought top and become active, the annotation engine parses the file, extracts all the Seahawk's annotations and notifies the view. In the notification, all the ids of the documents are reported. Through the query engine, the view retrieve the documents in order to display as well as it happens in the document navigator. Here, consistency could become a problem: Seahawk does not provide any mean to check the annotations put in the code. Thus, the annotations could link documents that do not exist anymore. As a result, there is no way to show such documents in the view, since the documents' data is unavailable.

To cope with this situation, we decided to show the document anyway, to add the message "[Not Available]" in front of the document's title and to make it not traversable. As well as the document navigator, also this view is coupled with the document's content view. The user can visualize part of the document and jump from one node to another. Differently from the document navigator view, D&D functionalities are disabled. Thus, the user cannot neither import code snippets nor create annotations from documents. The only functionalities provided concern the manipulation of annotations. Through a contextual menu, users can modify the comment of an annotation or delete the annotation as well. Of course, users can still directly manipulate the code and modify or delete annotations by hand, without having to use the view. Finally, users can access information regarding the link (*e.g.,* comment, author and creation time) through an overlay on the view that pops up when the mouse is over the document's title.

4.4.3 Document's Content View

As explained both in Section 4.4.1 and Section 4.4.2, the document's content view is a passive view that waits for documents, or documents' node, to be pushed into. The main purpose of the view is to render documents as an HTML page trough a web browser widget. The reason why we wanted to use HTML instead of building the view on top of the Eclipse IDE framework are mainly two.

Figure 4.8. The document's content view.

The former concerns the usability in terms of development, since using HTML results to be a more flexible solution from a rendering point of view. The latter concerns the freedom of defining a custom renderer to show different types of documents in the same environment. As we discussed in Section 4.2.3, every kind of document could be potentially represented. Another good feature that justifies the use of a web-browser in the view concerns the external links a developer can find in a document. For example, it is likely to have posts in Stack Overflow that refer to external resources (*e.g.,* Java documentation, forums, webpages *etc.*). Preventing users to access those resources brings to a limitation of usefulness of some documents. With the view we built, the user can navigate those resources and come back to the document.

In Figure 4.8 we depict the document's content view. It shows an example of Stack Overflow's document rendered in HTML. We have chosen a different layout to present the Stack Overflow's documents and we adopted this for all the Stack Exchange websites. For example, if the document is taken from another website different from Stack Overflow (*e.g.,* gamedev.stackexchange.com) the rendering would be the same. To render such documents we devised the following layout: We put a label on the left side of every posts, thus identifying is it is a question (Q) or an answer (A). Under the letter we put the score given by the user in Stack Overflow. Thanks to the structure of Q&A's content, we are able to easily identify code snippets by looking for the <*code*> tags. We used a Javascript library[6] to automatically put syntax highlighting on the text contained in the <*code*> tags, without knowing the programming language. To keep the view consistent with both the document navigator view and the suggested documents view, the coloring scheme for Q&A is the same: questions are orange, the accepted answer is green and the other answers are blue.

At the end of each post we put the author, date of creation and the possibility of visualizing comments. Differently from Stack Overflow, we decided to optionally show comments, thus following the Facebook[7] approach. A link reporting the number of comments for a specific post can be clicked by users, and the list of comments is then shown.

[6] `http://code.google.com/p/google-code-prettify/`
[7] `http://www.facebook.com`

4.4.4 Notification System

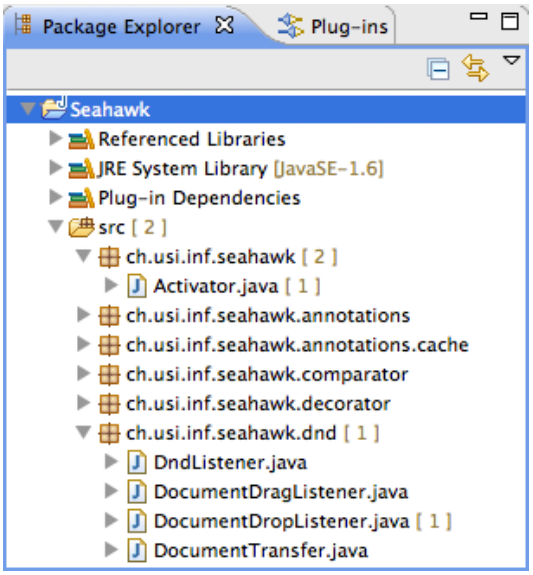


Figure 4.9. The package explorer notification system.

In Section 4.3.3 we discussed the core functionalities of the SEAHAWK plugin, that is, the annotations. Developers can put annotations in the code in a collaborative fashion and we also discussed the mechanism in the annotation engine that permits to understand what annotations were not previously linked to the code. Being part of the recommendation engine, we have to provide developers with a functionality that allows them to rapidly spot new annotations in the project.

To this aim we implemented a simple notification system embedded in the package explorer. An example of notification for a project is depicted in Figure 4.9. Whenever a project is refreshed, thus updating all the files contained in it, the annotation engine parses the files and create a list of annotations. Subsequently, it checks for all of them in the annotation cache to understand which one has not been seen yet by the developer. If some new annotations have been found, the notification system decorates the package explorer with the number of new annotation between square brackets. For example, in the image new annotations has been found for the compilation units *DocumentDropListener.java* and *Activator.java*. The related packages reports the cumulative count of the annotations (*e.g., ch.usi.inf.seahawk.dnd* reports 1). Since we treat packages as well as directories, the cumulative count is also present in root packages or folders. Therefore, the package *ch.usi.inf.seahawk* reports a total count of two as well as the root folder *src*. Whenever the developer opens one of the compilation units, the annotation engine parses the file and puts the annotations in the cache before the number shown in the package explorer is updated, thus reducing the count of the annotations.

4.4.5 Invoking SEAHAWK

In Section 4.3.2 we discussed how the query engine can automatically build queries given a program entity. We implemented two different way of interacting with this feature provided by the query engine: The former, as depicted in Figure 4.10, is implemented with a contextual menu.

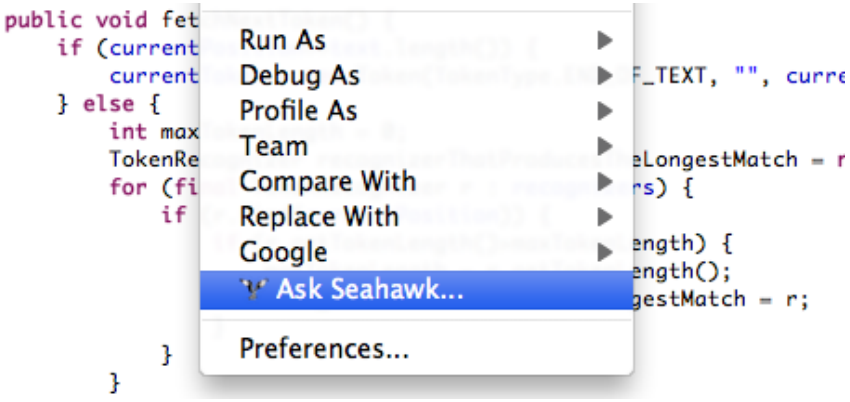


Figure 4.10. The contextual menu to invoke Seahawk.

A developer can get the *Ask Sehawk...* entry in the contextual menu from the Package Explorer, the Editor Outline and from the code editor. In the first two cases, the developer can click on a Java element represented in the tree (*e.g.,* class, method, package *etc.*) to access the menu and visualize Seahawk's entry. Of course, the selection is not restricted to just one element but can comprise many of them. If the developers clicks on an element which is not a Java Element (*e.g.,* a folder), this entry in the contextual menu becomes unavailable. In the last case, the developer can open the contextual menu by clicking on every part of the code. According to the position of the cursor, the query engine understands which is the referred Java element. When the developer invokes Seahawk, the set of relevant keywords is generated and put in the document navigator view as a query. The query is then automatically triggered.

The latter interaction is implement through key bindings available only in the code editor. A developer can thus invoke Seahawk by pressing Ctrl+U on the keyboard. As well as for the contextual menu, Seahawk understands the target entity by looking at the current position of the cursor in the text. Differently from the previous case, if the user invokes Seahawk from the keyboard, the query is not automatically fired. On the contrary, Seahawk just puts the keyowords in the query box inside the document navigator view and allows the developer to write some additional text. In doing so we let the user to specialize the query with some additional information.

Chapter 5

Evaluation

In this chapter we present a preliminary evaluation of SEAHAWK, without aiming at proving that these results are valid from a statistical point of view. We devised three experiments in which we tested the potential impact of the tool in border-line situations. We describe the approach we followed in those experiments and then, we conclude by discussing the results obtained, trying to asses the advantages and the drawbacks of the approach we followed in SEAHAWK.

5.1 Experiment I: Java Programming Exercises

The first experiment we devised is the one that really reaches the limit of the SEAHAWK' scope. In this task we wanted to assess to what extent our tool can deal with just text. To that aim, we used a set of exercises taken from a Java programming course [1] [2] and we tried to evaluate the relevance of the documents retrieved from the Stack Overflow's crowd knowledge by just extracting keywords from them.

According to what discussed so far in this thesis, we know that SEAHAWK is not designed to directly deal with just text. We had to recreate the right conditions to allow SEAHAWK to extract keywords from the exercise's text. Since it needs at least a Java entity, the only clean solution is to create a class stub and to put the entire exercise's text as comment before, or inside, the class body. We also consider the name of the Java entity as part of the keywords. To not bias the results, we gave a name that summarizes the topic of the exercise to the class.

Listing 5.1. Example of Java exercise prepared for the test

```
/* Write a class that implements the CharSequence interface
found in the java.lang package.
Your implementation should return the string backwards.
Select one of the sentences from this book to use as the data.
Write a small main method to test your class.
Make sure to call all four methods. */

public class CharSequenceImpl { }
```

[1] http://www.home.hs-karlsruhe.de/~pach0003/informatik_1/aufgaben/en/java.html
[2] http://codingbat.com/java

In Listing 5.1 we present one of the exercise that we tailored to let Seahawk correctly work. As we can see, the class is just a stub with no implementation. Since we used a compilation unit that contains just one class, every part of the code, including comments, is considered as part of the body of the class. We do not care about the additional words and characters introduced by this solution since they are either stop-words (*i.e., public* and *class*) or symbols to be discarded during the keywords extraction process (*i.e.,* curly brackets, slashes and stars).

With this environment we tested Seahawk on 35 exercises. For every exercise, we created a class similar to the one presented in the previous example, we generated keywords from it and we queried the search engine. From the result returned, we considered the first 15 documents. We manually inspected every document and we evaluated the relevance of the information contained in the discussion. With the term "relevant", we mean that the discussion can lead to a solution of the exercise either through the tackled topic or the code snippets. For example, let's take into consideration the exercise in Listing 5.1. If we find a discussion about the implementation of the *CharSequence* interface, thus showing what method must be implemented but tackling a complete different problem, we consider this as relevant. To avoid a binary notion of relevance (*e.g.,* useful, useless) we defined five levels of relevance, ranging from 0 to 4, to classify a document:

Relevance Level	Value
Highly Relevant	4
Relevant	3
Related	2
Slightly Related	1
Not Relevant	0

Table 5.1. Legenda: Document Relevance Levels

To have a numerical assessment of this experiment, we refer to the *normalized discounted cumulative gain* (NDCG). It is generally used to evaluate ranked retrieval results from search engines, by using a multi-valued notion of relevance [MRS08]:

$$NDCG(Q,k) = \frac{1}{\mid Q \mid} \sum_{j=1}^{|Q|} Z_{kj} \sum_{m=1}^{k} \frac{2^{R(j,m)} - 1}{log_2(1+m)} \tag{5.1}$$

where k is the size of the result set, Q is the set of queries performed, *R(j,d)* is the relevance score gave to document d for query j and Z_{kj} is the normalization factor calculated such that $NDCG = 1.0$ in the ideal scenario (*i.e.,* all the documents have the maximum level of relevance). In our experiment, $k = 15$, $|Q| = 35$ and the normalization factor we calculated is $Z_{kj} = 0.011373948$.

5.1.1 Experiment I: Discussion

In the first experiment we wanted to asses the behavior of Seahawk in dealing with text. We wanted to have a numerical assessment of the experiment and we used the *NDCG* value for that purpose. However, the result we obtained from the index is 9.07%, thus meaning that barely one time in ten the documents retrieved were relevant for the Java exercise we used. In Table 5.2 and Table 5.3 we present a subset of the data collected. For the complete data, we refer to Table A.5 and Table A.6 in Appendix A.

Exercise Name	D1	D2	D3	D4	D5	D6	D7	D8
ElectricalResistance	0	0	0	0	0	0	0	0
Fibonacci	2	3	3	0	0	0	2	3
Metropolis	0	0	0	0	0	0	0	0
NaturalMergeSort	3	3	4	0	3	0	4	3
RouletteStrategy	0	0	0	0	0	0	0	0
SudokuSolver	3	4	3	2	0	0	0	0
WindSpeed	0	0	0	0	0	0	0	0

Table 5.2. Experiment I Data: Part I *(0 = Not Relevant, 4 = Highly Relevant)*

Exercise Name	D9	D10	D11	D12	D13	D14	D15
ElectricalResistance	0	0	0	0	0	0	0
Fibonacci	3	3	3	3	3	3	4
Metropolis	0	0	0	0	0	0	0
NaturalMergeSort	0	0	0	3	2	2	2
RouletteStrategy	0	0	0	0	0	0	0
SudokuSolver	0	0	0	0	0	0	1
WindSpeed	0	0	0	0	0	0	0

Table 5.3. Experiment I Data: Part II *(0 = Not Relevant, 4 = Highly Relevant)*

Even though the value of the *NDCG* is not encouraging, there are some considerations to take into account. First of all, we noticed that the SEAHAWK's approach failed on exercises being too much simple. For example, exercises like *ElectricalResistance* or *WindSpeed*, where the student is asked to write a simple function to calculate the value of the resistance and the wind speed value, provided too few information and the document returned were not related at all. Sometimes the topic of the exercise was a subset of a more complex one. For instance, *RouletteStrategy* required to calculate the number of turns required to lose all by betting betting only on color at poker. The documents retrieved for this exercise were discussing the same topic but with an higher level of difficulty (*e.g.,* machine learning approach), thus making them not relevant at all.

One reason why the results for those exercises were not relevant could reside in the absence of of information in Stack Overflow's crowd knowledge. Even though Stack Overflow has a lot of discussions regarding homework, the requirements of the exercises were not enough specific. Just in one case the exercise was in one of the document returned. Another reason concerning the difficulties on retrieving relevant documents could reside in exercises requiring the implementation of data-classes (*e.g., Metropolis*). The documents returned for those topics were completely not related. When the exercise tackled a well known topic, the relevance of the documents increased. We were able to find out solutions or even the implementations in pseudocode, Java or similar languages that could be easily adapted to Java. For example, exercises requiring the implementation of the fibonacci algorithm, sorting algorithms or sudoku solvers (*i.e., Fibonacci*, *NaturalMergeSort* and *SudokuSolver*) returned an high number of relevant documents that could have provided the solution.

5.2 Experiment II & III: Method Stubs and Method Bodies

The second and the third experiments concerned the evaluation of Seahawk in other two borderline contexts. We wanted to assess the impact of Seahawk in dealing methods stubs and fully implemented methods. We decided to use real implementations taken from projects and, in some cases, we implemented simple programs to solve basic exercises.

For the second experiment, we selected eight different methods in projects developed by students, and two exercises taken from a Java programming course, thus reaching a total of ten methods where 50% of the methods were implementing part of a Java interface while the remaining 50% were random methods. This distribution is done on purpose since we wanted to see if the behavior in case of interface changes in respect to other types of method. Differently from the first experiment, we have not changed or tailored the code, but we just removed the implementation from the target method to obtain a stub, thus leaving everything else unchanged.

For the third experiment, we selected seven methods with full implementation. Only two methods were implementing an interface. Differently from the previous experiments, this one has been done to assess the behavior of Seahawk while having the full solution at hand. We wanted to see if the documents returned by the search engine could have been used to help the developer during the development phase. The assessment of the documents for both the experiments takes place as well as for the first one. However, we do not calculate the *NDCG* but we limited to observe and assess the documents.

5.2.1 Experiment II & III: Discussion

Due to the limited amount of data, we cannot use any index to get a numerical assessment of the Seahawk's behavior. However, we believe that the *NDCG* would not have been improved in respect to the first experiment.

Listing 5.2. Example of Java method stub

```
@Override
public boolean addAll(Collection<? extends Integer> arg0) {
    return false;
}
```

While dealing with method stubs, Seahawk has mainly two different behaviors. Before describing the insights we have regarding the behavior of the tool in such a context, we have to make a premise: A method stub can provide just the name of the method and the potential tokens contained in the imports. For example, the stub in Listing 5.2 would provide the query "Collection arg addAll override java addall collection add util", where tokens such as *java*, *util* and *collection* are provided by the import *java.util.collection.Collection*, while the remaining are extracted from the method. All the keywords generated from the stub refer to the name of the method or to the parameters. The scope of such queries are limited to the minimal information provided by it.

Seahawk brought to relevant results in case the stub concerning an interface method. Due to its unchangeable signatures, methods implementing interfaces could provide enough information to get relevant documents. For instance, the *decorate* method had really useful documents in the first entries. This method is a method interface used in the Eclipse framework to implement a package explorer decorator. By reading the documents returned, a developer could have obtained the right examples to implement a fully working decorator (*i.e., REmailLightweightDecorator (decorate)*).

Type	Class (MethodName)	D1	D2	D3	D4	D5	D6	D7	D8
I	REmailLightweightDecorator (decorate)	4	2	4	0	0	0	0	0
NI	SpreadsheetReader (removeDoubleQuotes)	0	2	1	1	0	1	0	0

Table 5.4. Experiment II & III Data: Part I
(0 = Not Relevant, 4 = Highly Relevant, I = Interface, NI = Not Interface)

Type	Class (MethodName)	D9	D10	D11	D12	D13	D14	D15
I	REmailLightweightDecorator (decorate)	0	0	0	0	0	0	0
NI	SpreadsheetReader (removeDoubleQuotes)	0	1	0	0	0	0	0

Table 5.5. Experiment II & III Data: Part II
(0 = Not Relevant, 4 = Highly Relevant, I = Interface, NI = Not Interface)

However, this does not apply in case of random methods. The documents retrieved were completely irrelevant unless the method's name contained some keywords regarding a very specific topic. For instance, the method's name *removeDoubleQuotes* (*i.e., SpreadsheetReader (removeDoubleQuotes)*), pointed out discussion regarding regular expressions that, in this specific case, could have led to a better solution in respect to the one implemented in the full method. In Table 5.4 and Table 5.5 we report the results for the two aforementioned stubs.

The difference in the Seahawk's behavior changed while dealing with fully implemented methods. Due to additional information provided by the body, having an interface method did not affected the results. On the contrary, we believe that some other factors influenced the outcome of the research. First of all, the single responsibility principle could play an important role: If a method has its own delimited scope and accomplishes one single goal, the resulting number topics would be limited as well. Thus, the scope of the query could be more specific and could led to better results. Secondly, the library or the framework used in the implementation of the method can refine the scope of the research. For example, a method implementing a menu on top of the SWT framework should better define the scope of the query. Somehow, the classes used play the role of the topic in the method and the more popular those classes are, the highest the chance of getting relevant results. We do not have any evidence to sustain this idea, but it is just a feeling we had by looking at the results gathered. For a complete overview of the data gathered in this two experiments we refer to Table A.3, Table A.4, Table A.1, and Table A.2 in Appendix A.

Chapter 6

Conclusions

As we stated in Section 1.1, the contributions of this thesis are:

- Stack Overflow crowd knowledge integration in the IDE.
- Linkage between relevant discussions and source code through an ad-hoc annotations system.
- Sub-versioning system integration to provide developers with collaborative support, exploiting valuable Q&A in the context of team work.
- Example-based programming by allowing developers to retrieve code snippets from Stack Overflow Q&A directly in the code editor.
- Automated creation of queries from source code entities.

In the following, we discuss how we tackled those points in this thesis, through the implementation of our approach. As we initially discussed in Chapter 3, we believe that integrating the crowd knowledge in the IDE would be a valuable support for developers. In Chapter 4 we presented the implementation of the aforementioned approach by presenting Seahawk , a tool for the Eclipse IDE. With Seahawk we showed how Q&A services and, in particular, the Stack Overflow's crowd knowledge can be integrated in the IDE by mean of a recommendation system. We then presented the three main components of Seahawk: the data-collection mechanism, the recommendation engine and the user interface.

In Section 4.2 we discussed the approach we followed to gather data from Q&A services. We described the architecture of the data-collection mechanism, the pro and the cons of our solution, and the the design idea behind the document model. In Section 4.3 we presented the key idea behind the recommendation engine. We explained how we provided suggestions regarding Stack Overflow's documents by implementing a manual and automatic interaction with the plugin, the rational behind the query engine implemented in Seahawk. We also showed how the tool can provide both an usual manual interaction with a search engine as well as it can produce queries for it by analyzing the code entities written by the developer. We described the approach for the collaborative support: with an annotation system, developers can link Stack Overflow's documents to a version of a source file by transparently integrating versioning systems.

Finally, in Section 4.4 we presented the user interface of Seahawk. We discussed the different views and the approach followed to let developers navigate documents. We also explained how we used drag&drop interactions to let developers import code snippets from documents to the code editor, thus favoring example-based programming in a language independent fashion.

6.1 Future Work

We showed how we achieved the contributions stated in Section 1.1 by implementing our recommendation system: Seahawk. We discussed how Seahawk and, in particular the plugin side, provided developers with the needed features to leverage the Q&A services in the software development process. Even though we can claim that those features meet the requirements posed by the contributions we stated, there are still enhancements to be done:

- We can improve Seahawk by making it less coupled with Apache Solr. A future work concerns the definition and construction of an intermediate web service between the search engine and the plugin. The introduction of such a web service would allow new solutions regarding the processing of the queries made by Seahawk. We designed the Apache Solr's document schema to contain the information regarding the tags. We do not use of such a field but we just defined it just for further development. In particular, the combination of the information present in the field and the web service in front of the search engine can allow us to provide tag filtering on the queries, thus having the same feature as the Stack Overflow's search engine provides. The web service could periodically crawl the Stack Exchange API to get the list of tags. Any query coming from the Seahawk plugin could be parsed to check the presence of tags in it, thus refining the scope of the actual query.

- The user interface of Seahawk can be enhanced. Instead of using a drag&drop interaction to import code snippets, we could provide an ad-hoc dialog to be called inside the code editor. This solution would save space in the Eclipse workspace (there would be no more need of having a tab always opened) providing the same navigation system for documents.

- A valuable improvement is to provide a Q&A service inside the context of project. Seahawk could be provided with additional features in its backbone to allow developers to ask questions to teammates. This would help newcomers to understand some part of the project by asking questions to more experienced teammates. The discussions stored would build a knowledge that could also document the evolution of the system.

Seahawk provides useful features to exploit the crowd knowledge of Q&A services. The improvements discussed in this section can bring our application to a higher level of efficiency. Even though our approach is still not perfect, we believe that it is a valuable way to leverage and integrate crowd knowledge in the software development process.

Appendix A

Experimental Data

In this appendix we present the data referring to the evaluation of Seahawk discussed in Chapter 5. All the following tables summarize the evaluations of the documents retrieved for a specific task in each experiment.

Class (MethodName)	D1	D2	D3	D4	D5	D6	D7	D8
REmailLightweightDecorator (decorate)	0	0	0	0	0	0	0	0
SpreadsheetReader (removeDoubleQuotes)	0	0	0	0	0	0	0	0
Parser (parseFunction)	0	0	0	0	0	0	0	0
SpreadsheetReader (loadFile)	2	1	2	0	0	0	4	4
MarkerInitActionDelegate (prepareSQLite)	2	3	0	2	0	2	0	0
MapEditor (buildMenu)	1	0	4	4	4	4	0	0
MapEditor (buildWest)	2	0	2	0	2	2	1	3

Table A.1. Method Bodies: Experiment Data Part I *(0 = Not Relevant, 4 = Highly)*

Class (MethodName)	D9	D10	D11	D12	D13	D14	D15
REmailLightweightDecorator (decorate)	3	0	0	0	0	0	0
SpreadsheetReader (removeDoubleQuotes)	0	0	0	0	0	0	0
Parser (parseFunction)	0	0	0	0	0	0	0
SpreadsheetReader (loadFile)	2	2	0	0	0	4	0
MarkerInitActionDelegate (prepareSQLite)	0	0	0	0	0	0	0
MapEditor (buildMenu)	0	0	0	0	1	4	1
MapEditor (buildWest)	3	2	0	0	0	2	0

Table A.2. Method Bodies: Experimental Data Part II *(0 = Not Relevant, 4 = Highly)*

Type	Class (MethodName)	D1	D2	D3	D4	D5	D6	D7	D8
I	EnumerationImpl (hasMoreElements)	0	0	3	4	0	0	4	0
I	REmailLightweightDecorator (decorate)	4	2	4	0	0	0	0	0
I	IntegerList (addAll)	2	2	2	3	1	0	2	0
I	MarkerInitActionDelegate (selectionChanged)	0	0	0	0	0	0	0	0
I	PreferencePaneMbox (createFieldEditors)	0	1	0	0	0	2	0	0
NI	MarkerInitActionDelegate (prepareSQLite)	0	0	0	0	0	0	0	0
NI	CopyPaste (copy)	0	0	0	0	0	0	0	0
NI	SpreadsheetReader (loadFile)	0	0	0	0	0	0	0	0
NI	SpreadsheetReader (removeDoubleQuotes)	0	2	1	1	0	1	0	0
NI	Parser (parseFunction)	0	0	0	0	0	0	0	0

Table A.3. Method Stubs: Experimental Data Part I
(0 = Not Relevant, 4 = Highly Relevant, I = Interface, NI = Not Interface)

Type	Class (MethodName)	D9	D10	D11	D12	D13	D14	D15
I	EnumerationImpl (hasMoreElements)	0	0	0	0	0	0	0
I	REmailLightweightDecorator (decorate)	0	0	0	0	0	0	0
I	IntegerList (addAll)	0	1	0	0	1	2	2
I	MarkerInitActionDelegate (selectionChanged)	0	0	0	0	0	0	0
I	PreferencePaneMbox (createFieldEditors)	0	0	0	0	0	0	0
NI	MarkerInitActionDelegate (prepareSQLite)	0	0	0	0	0	0	0
NI	CopyPaste (copy)	0	0	0	0	0	0	0
NI	SpreadsheetReader (loadFile)	0	0	0	0	1	1	1
NI	SpreadsheetReader (removeDoubleQuotes)	0	1	0	0	0	0	0
NI	Parser (parseFunction)	0	0	0	0	0	0	0

Table A.4. Method Stubs: Experimental Data Part II
(0 = Not Relevant, 4 = Highly Relevant, I = Interface, NI = Not Interface)

Exercise Name	D1	D2	D3	D4	D5	D6	D7	D8
Metropolis	0	0	0	0	0	0	0	0
UnicodeToUTF8	0	0	0	0	0	0	0	0
ElectricalResistance	0	0	0	0	0	0	0	0
ChemicalElements	0	0	0	0	0	0	0	0
WindSpeed	0	0	0	0	0	0	0	0
RationalNumber	0	0	0	0	0	0	0	0
Polynomial	0	0	0	2	2	2	0	0
PrintTable	0	0	0	0	0	0	0	0
SortThreeNumber	0	0	0	0	0	0	0	0
GermanGrades	0	0	0	0	0	0	0	0
DayOfWeek	0	0	0	0	0	0	0	0
RouletteStrategy	0	0	0	0	0	0	0	0
NumberOfBytes	0	0	0	0	0	0	0	0
SimpleCalculator	0	0	0	0	0	0	2	2
Anagrams	0	4	0	0	0	0	0	3
SumOfDigits	0	1	1	0	0	0	0	0
SmallNeighboringDistance	0	0	1	0	0	0	0	0
SudokuSolver	3	4	3	2	0	0	0	0
SudokuCreator	1	1	1	0	1	1	1	0
EnglishPegSolitaireSolver	3	3	0	2	1	2	0	0
ZerosOfContinuosFunction	0	0	0	0	0	0	0	0
NewtonMethod	0	0	3	0	0	0	0	0
BottomUpMergeSort	1	1	0	1	0	0	0	0
ShellSort	0	1	3	0	0	0	0	0
NaturalMergeSort	3	3	4	0	3	0	4	3
Queue	2	0	0	0	0	3	0	0
CompareChemicalElements	3	1	0	0	0	0	0	2
HexToDecimal	0	0	0	0	0	0	0	0
LoadChemicalElementsFromFile	0	0	0	0	0	0	0	0
Fibonacci	2	3	3	0	0	0	2	3
Factorial	3	0	0	0	0	4	3	0
PowerN	4	4	4	0	0	0	0	0
GroupSum	0	1	1	0	0	0	0	1
CountCode	0	0	1	0	0	0	0	0
CharSequenceImpl	0	3	2	4	2	1	1	0

Table A.5. Java Programming Exercises: Experimental Data - Part I
(0 = Not Relevant, 4 = Highly Relevant)

Exercise Name	D9	D10	D11	D12	D13	D14	D15
Metropolis	0	0	0	0	0	0	0
UnicodeToUTF8	0	0	0	0	0	0	1
ElectricalResistance	0	0	0	0	0	0	0
ChemicalElements	0	0	0	0	0	0	0
WindSpeed	0	0	0	0	0	0	0
RationalNumber	0	0	0	0	4	2	0
Polynomial	0	0	0	0	1	2	1
PrintTable	1	0	0	0	0	0	0
SortThreeNumber	0	0	0	0	0	0	0
GermanGrades	0	0	0	0	0	2	0
DayOfWeek	0	0	0	0	0	0	0
RouletteStrategy	0	0	0	0	0	0	0
NumberOfBytes	0	0	0	0	0	0	0
SimpleCalculator	0	4	3	4	0	0	0
Anagrams	0	0	0	2	0	4	0
SumOfDigits	0	0	0	0	0	0	0
SmallNeighboringDistance	0	0	0	0	0	0	0
SudokuSolver	0	0	0	0	0	0	1
SudokuCreator	0	1	4	1	1	0	0
EnglishPegSolitaireSolver	0	0	0	0	0	0	0
ZerosOfContinuosFunction	3	0	0	0	0	0	0
NewtonMethod	0	0	0	0	4	3	3
BottomUpMergeSort	1	1	0	0	1	0	0
ShellSort	0	0	0	0	0	0	0
NaturalMergeSort	0	0	0	3	2	2	2
Queue	0	0	0	0	0	0	0
CompareChemicalElements	4	0	0	0	0	0	0
HexToDecimal	0	1	0	0	0	0	0
LoadChemicalElementsFromFile	0	0	0	0	0	0	0
Fibonacci	3	3	3	3	3	3	4
Factorial	4	0	0	0	0	0	4
PowerN	0	0	0	0	0	0	0
GroupSum	1	1	0	0	1	1	0
CountCode	0	0	0	0	0	0	2
CharSequenceImpl	0	0	0	1	0	1	0

Table A.6. Java Programming Exercises: Experimental Data - Part II
(0 = Not Relevant, 4 = Highly Relevant)

Bibliography

[AZBA08] Lada A Adamic, Jun Zhang, Eytan Bakshy, and Mark S Ackerman. Knowledge sharing and yahoo answers: everyone knows something. In *WWW '08: Proceeding of the 17th international conference on World Wide Web*. ACM, April 2008.

[BCLM11] Alberto Bacchelli, Anthony Cleve, Michele Lanza, and Andrea Mocci. Extracting structured data from natural language documents with island parsing. In *ASE '11: Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, November 2011.

[BDWK10] Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R Klemmer. Example-centric programming: integrating web search into the development environment. In *CHI '10: Proceedings of the 28th international conference on Human factors in computing systems*. ACM Request Permissions, April 2010.

[BLH11] Alberto Bacchelli, Michele Lanza, and Vitezslav Humpa. RTFM (Read the Factual Mails) - Augmenting Program Comprehension with Remail. *CSMR*, pages 15–24, 2011.

[BLR10] Alberto Bacchelli, Michele Lanza, and Romain Robbes. Linking e-mails and source code artifacts. In *ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*. ACM Request Permissions, May 2010.

[BPL12] Alberto Bacchelli, Luca Ponzanelli, and Michele Lanza. Harnessing Stack Overflow for the IDE. *In Proceedings of RSSE 2012 (3rd International Workshop on Recommendation Systems for Software Engineering)*, April 2012.

[ČMSB04] Davor ČubraniĆ, Gail C Murphy, Janice Singer, and Kellogg S Booth. Learning from project history: a case study for software development. In *CSCW '04: Proceedings of the 2004 ACM conference on Computer supported cooperative work*. ACM Request Permissions, November 2004.

[Cor89] Thomas A Corbi. Program Understanding: Challenge for the 1990s. *IBM Systems Journal ()*, 28(2):294–306, 1989.

[GK10] Florian S Gysin and Adrian Kuhn. A trustability metric for code search based on developer karma. In *SUITE '10: Proceedings of 2010 ICSE Workshop on Search-driven Development: Users, Infrastructure, Tools and Evaluation*. ACM Request Permissions, May 2010.

[GM09] M. Goldman and R.C. Miller. Codetrail: Connecting source code and web resources. *Journal of Visual Languages & Computing*, 20(4):223–235, 2009.

[HAM10] Sonia Haiduc, Jairo Aponte, and Andrian Marcus. Supporting program comprehension with source code summarization. In *ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*. ACM Request Permissions, May 2010.

[HAMM10] Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. On the Use of Automated Text Summarization Techniques for Summarizing Source Code. In *WCRE '10: Proceedings of the 2010 17th Working Conference on Reverse Engineering*. IEEE Computer Society, October 2010.

[Hat10] M McCandless E Hatcher O Hatcher. *Lucene in Action, Second Edition: Covers (text only) 2nd(Second) edition by M.McCandless.E.Hatcher.O.Hatcher*. Lucene, Jmeter, Apache Http Server, Apache Tomcat, Nutch, Spring Framework, Jspwiki, Codeigniter. Manning Publications;, 2 edition edition, 2010.

[HB08] Reid Holmes and Andrew Begel. Deep intellisense: a tool for rehydrating evaporated information. In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*. ACM, May 2008.

[HP00] Morten Hertzum and Annelise Mark Pejtersen. The information-seeking practices of engineers: searching for documents as well as for people. *Information Processing and Management: an International Journal*, 36(5), September 2000.

[KDV07] Andrew J Ko, Robert DeLine, and Gina Venolia. Information Needs in Collocated Software Development Teams. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, May 2007.

[KELN10] Adrian Kuhn, David Erni, Peter Loretan, and Oscar Nierstrasz. Software Cartography: thematic software visualization with consistent layout. *Journal of Software Maintenance ()*, 22(3):191–210, 2010.

[KM06] Mik Kersten and Gail C Murphy. Using task context to improve programmer productivity. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*. ACM Request Permissions, November 2006.

[LR93] Robert Lougher and Tom Rodden. Supporting Long-term Collaboration in Software Maintenance. In *COCS '93: Proceedings of the conference on Organizational computing systems*, pages 228–238, New York, New York, USA, 1993. ACM Press.

[LS81] Bennet P Lientz and E Burton Swanson. Problems in Application Software Maintenance. *Commun. ACM ()*, 24(11):763–769, 1981.

[LVD06] Thomas D LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models: a study of developer work habits. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*. ACM Request Permissions, May 2006.

[MMM+11] L. Mamykina, B. Manoim, M. Mittal, G. Hripcsak, and B. Hartmann. Design lessons from the fastest q&a site in the west. *Proceedings of the 2011 annual conference on Human factors in computing systems*, pages 2857–2866, 2011.

[MRS08] Christopher D Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 1 edition, July 2008.

[NAA09] Kevin Kyung Nam, Mark S Ackerman, and Lada A Adamic. Questions in, knowledge in?: a study of naver's question answering community. In *CHI '09: Proceedings of the 27th international conference on Human factors in computing systems*. ACM Request Permissions, April 2009.

[Ras00] Jef Raskin. *The Humane Interface: New Directions for Designing Interactive Systems*. Addison-Wesley Professional, April 2000.

[RWZ10] Martin P Robillard, Robert J Walker, and Thomas Zimmermann. Recommendation Systems for Software Engineering. In *ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 80–86. ACM Request Permissions, July 2010.

[SHM+10] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K Vijay-Shanker. Towards automatically generating summary comments for Java methods. In *ASE '10: Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM Request Permissions, September 2010.

[SM11] Nicholas Sawadsky and Gail C Murphy. Fishtail: from task context to source code examples. In *TOPI '11: Proceedings of the 1st Workshop on Developing Tools as Plug-ins*. ACM Request Permissions, May 2011.

[STvDC10] M.A. Storey, C. Treude, A. van Deursen, and L.T. Cheng. The impact of social media on software engineering practices and tools. *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 359–364, 2010.

[TBS11] Treude, Christoph, Ohad Barzilay, and Margaret-Anne Storey. How do programmers ask and answer questions on the web? (NIER track). In *ICSE '11: Proceeding of the 33rd International Conference on Software Engineering*. ACM Request Permissions, May 2011.

[Tre12] Treude, Christoph. Programming in a Socially Networked World: the Evolution of the Social Programmer. pages 1–3, January 2012.

[Ye06] Yunwen Ye. Supporting software development as knowledge-intensive and collaborative activity. In *WISER '06: Proceedings of the 2006 international workshop on Workshop on interdisciplinary software engineering research*. ACM, May 2006.