Leveraging Crowd Knowledge for Software Comprehension and Development

Luca Ponzanelli, Alberto Bacchelli, Michele Lanza
REVEAL @ Faculty of Informatics – University of Lugano, Switzerland

***Abstract*—Question and Answer (Q&A) services, such as Stack Overflow, rely on a community of programmers who post questions, provide and rate answers, to create what is termed "crowd knowledge". As a consequence, these services archive voluminous and potentially useful information to help developers to solve programming-specific issues. Programmers tap into this crowd knowledge through web browsers. This requires them to step out of their integrated development environments (IDE), formulate a query, inspect the returned results and manually port the solution back to the IDE.**

We present an integrated and largely automated approach to assist programmers who want to leverage the crowd knowledge of Q&A services. We give a form to our approach by implementing SEAHAWK, an Eclipse plugin. SEAHAWK automatically formulates queries from the current context in the IDE, and presents a ranked and interactive list of results. SEAHAWK lets users identify individual discussion pieces and import code samples through simple drag & drop. Users can also link Stack Overflow discussions and source code persistently. We performed an evaluation of SEAHAWK, with promising results.

***Keywords*-Q&A services, recommendation systems**

I. INTRODUCTION

Software developers are continuously introduced to new technologies, components, and ideas [1]. New technologies are used to develop new components, but can also be used to provide developers with tools for software maintenance—an important part of software development [2] [3]. One reason why maintenance is difficult and time-consuming is that project documentation is hard to link to actual maintenance tasks [4]. Thus, developers need to ask questions to other programmers or teammates [5][6] and spend considerable time to obtain the desired information [7]. Among the available online resources, Q&A services provide developers with the infrastructure to exchange knowledge in form of questions and answers: Developers pose questions and receive answers regarding issues from people that are not part of the same project. Even though researchers pointed out that Q&A services could not provide high level technical answers [8] [9] [10], these services are filling "archives with millions of entries that contribute to the body of knowledge in software development" and they often become the substitute of the official product documentation [11].

A prominent example of technical Q&A service is Stack Overflow[1]. Mamykina *et al.* reported that it has gained popularity among developers and is becoming an important venue for sharing knowledge on software development.

[1]http://stackoverflow.com

On Stack Overflow more than 92% of the questions on expert topics are answered in a median time of 11 minutes [10]. Treude *et al.* pointed out how "Stack Overflow is particularly effective for code reviews, for conceptual question and for novices" [11]. Even tough they also pointed out how Stack Overflow is aimed at "a general audience that is not part of the same project", open source projects, such as Aptana[2], use Stack Overflow as a project documentation means.

Despite the knowledge provided by Q&A services, it cannot be leveraged from within an integrated development environment (IDE). Developers spend most of their time in the IDE to write and understand code [4] and they should be only focused on the current task without any major interruption or disturbance [12]. However, developers are forced to leave the IDE, thus interrupting the programming flow and lowering their focus on the current task. To reduce costs of modifying and maintaining large systems, by also improving the understanding of programs, tool support is needed [3]. Recommendation systems [13] are one form of such tools: According to Robillard *et al.*,"recommendation systems for software engineering (RSSE) are emerging to assist developers in various activities, from reusing code to writing effective bug reports" [1]. RSSEs play the role of personal assistants that can guide the programmer while developing or maintaining a software system by providing additional information. This information can be gathered from the crowd knowledge provided by Q&A services.

We make the following contributions. We present SEAHAWK[3] [14], a recommendation system in the form of a plugin for the Eclipse IDE[4] to harness the crowd knowledge of Stack Overflow from within the IDE. SEAHAWK mines the Stack Overflow knowledge base, displays the search results directly in the IDE, allows developer to link discussions to code entities and to import code snippets, and also offers the possibility to automatically generate queries by extracting keywords from the code entities given in the IDE. To evaluate the approach implemented in SEAHAWK we present and discuss a series of experiments.

Structure of the paper. In Section II we present SEAHAWK. In Section III we illustrate its usage with a scenario and present an evaluation in Section IV. In Section V we discuss related work. In Section VI we draw our conclusions.

[2]http://www.aptana.com/
[3]http://seahawk.inf.usi.ch
[4]http://eclipse.org

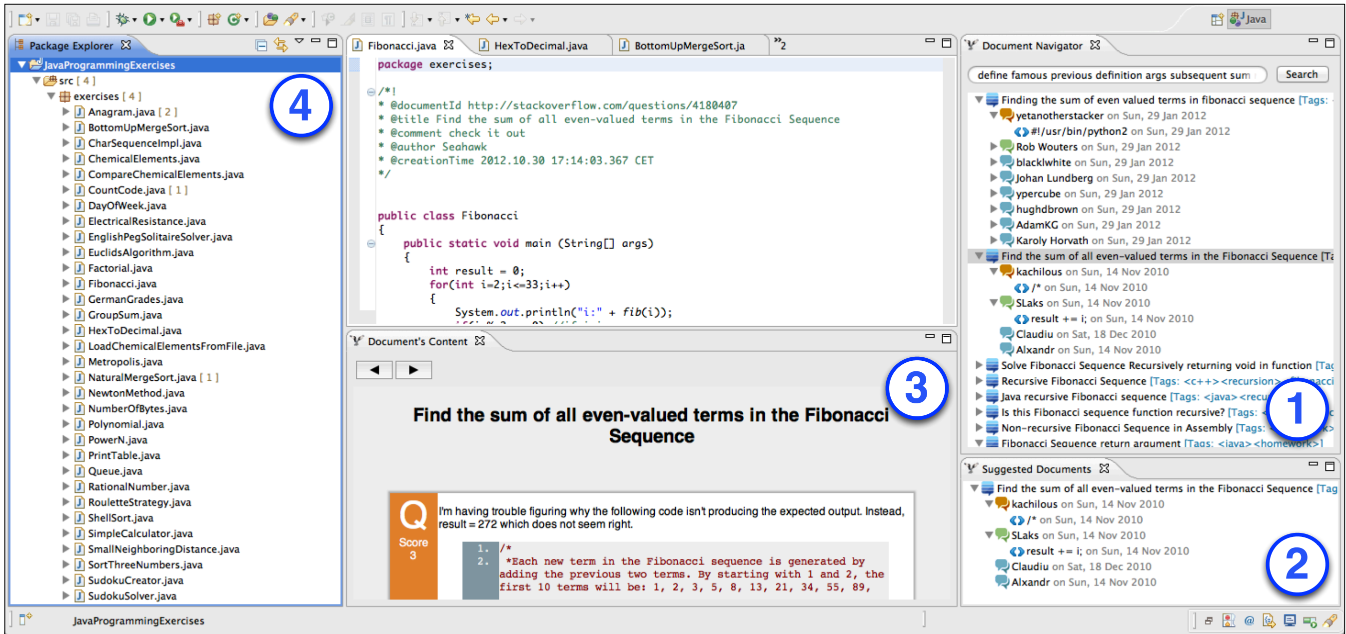

Figure 1: Seahawk User Interface

II. Seahawk

Figure 1 depicts the user interface of Seahawk. Users are provided with four main components to interact with Seahawk: (1) *Document Navigator View*, where the user can type in a query and navigate the returned documents, (2) *Suggested Document View*, where Seahawk suggests documents that are linked by the annotations in the code, (3) *Document's Content View*, where the content of the current document is presented to the user, and (4) a notification system inside the package explorer to notify developers of new linked documents. We refer to [15] for additional details.

In the following we present the architecture of Seahawk according to the structure of a recommendation system defined by Robillard *et al.* [1]: *A data-collection mechanism*, a *recommendation engine* and a *user interface*.

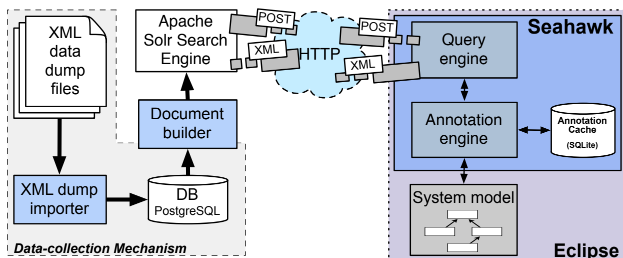

Figure 2: Seahawk's architecture

A. The Architecture

Figure 2 depicts Seahawk's architecture. The first component in Seahawk is the data collection mechanism, which is responsible for gathering Q&A data from Stack Overflow. We import Stack Overflow documents from a public data dump provided as local XML files. The data is extracted through a *XML dump importer* and stored in a relational database for performance reasons. We built a tool to query the database and to build a JSON representation of each document (thus making it available for any language). This representation is then included in an additional document schema required by the Apache Solr[5] search engine. When documents are indexed by the search engine, they become available for query. Apache Solr provides a RESTful interface to perform searches by means of GET and POST requests and it replies with XML data with the relevant documents.

According to Robillard *et al.*, interaction with recommendation systems can be both manual (*i.e.*, a query is inserted by the user) and automatic (*i.e.*, the recommendation engine generates the query) [1]. Seahawk supports both: users can manually write queries to retrieve documents, or Seahawk can extract keywords from code entities, build a query, and suggest documents. Seahawk also provides an annotation system that allows developers to link Stack Overflow documents to the source code.

[5] http://lucene.apache.org/solr/

B. Data Collection Mechanism

Stack Exchange offers a public RESTful API[6] to access the Stack Overflow content. Since the API is limited in usage and search capabilities, we decided to use the public data dump[7] provided by Stack Exchange, which comprises several XML files that represent the database of each website. We limit the files needed to the ones representing the data (*i.e., posts.xml, users.xml, comments.xml*), discarding the files regarding the evolution of the website (*e.g., posthistory.xml, badges.xml, votes.xml*), since we are not interested in data regarding the interactions of the users with the community.

On the left hand side of Figure 2, we depict the process to import and manipulate the data to reconstruct documents indexed by the search engine. We consider three XML files: *posts.xml, users.xml, comments.xml*. The total amount of entries in *posts.xml* sums up to more than 7 millions. To recreate a document, we need to gather a question and all the related answers from *posts.xml*. For the opening question and all answers, we extract information regarding users (*users.xml*), comments (*comments.xml*) and authors (*users.xml*). Since performing these operations by manipulating data directly from the XML files is resource intensive, we import everything in a database (thus also easing the document extraction). We chose to represent documents in JSON format to make them portable. To build documents, we implemented an importer that queries the database. Those documents are then included in an additional document representation and indexing required by Apache Solr.

The Search Engine

The search engine indexes documents when extracted and reconstructed from the database, and makes them available for queries. We take advantage of Apache Solr which stores and indexes documents in a vector space model, relying on Apache Lucene[8] as core engine. The weighting algorithm used by Apache Lucene, and thus by Apache Solr, is a variation of the standard *tf-idf* [16]. We configured Apache Solr to remove stop words, filter out possessive words, stem words, trim white spaces, filter synonyms and lower the case (see [17]) at both query and indexing time.

Once the indexing phase is complete, the Apache Solr engine can be queried via HTTP in a RESTful fashion. SEAHAWK can thus query the search engine to get relevant documents in XML format that contain the JSON representation of the original ones. The documents are then deserialized and shown in the Eclipse IDE.

C. The Recommendation Engine

The recommendation engine of SEAHAWK provides both manual and automatic interactions. The core is composed of a query engine and an annotation engine.

[6]https://api.stackexchange.com
[7]http://www.clearbits.net/creators/146-stack-exchange-data-dump
[8]http://lucene.apache.org/

The Query Engine

SEAHAWK's Eclipse plugin makes the Q&A crowd knowledge available in the IDE. Users can interact with this knowledge in ways that the website normally does not allow, such as directly manipulating code snippets. The main goal of the query engine is to communicate with Apache Solr, by creating a query given an input string. Being Q&A documents the target of such queries, it is likely to have some information also in the title, that is, the question itself. We assign more weight to the document's title to exploit possible keywords that can be relevant for the target search. Let us assume that a developer wants to query the search engine with the following query: "change label color in Java". The query engine tokenizes the string inserted by the developer. 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.

In the 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 given query (*e.g.,* titles containing words such as *label* or *color*) are retrieved even if the document's body does not match any of the tokens.

Automation of Queries

The query engine also provides an automatic keyword extraction feature to build queries. The first technical issue to overcome regards the code written by developers. Developers need to understand their code even though it does not compile. Dealing with code that does not compile has drawbacks: Compiling code can provide a full Abstract Syntax Tree (AST), but with compilation errors the AST can be partial or even absent. Moreover, the partial AST is the representation of the code until the compilation failed, thus discarding any additional information that comes after. This also applies for Eclipse when it is asked to produce an AST for a Java program.

To overcome this problem we use island parsing [18]. It copes with code that does not compile, to identify structural information of code entities (*i.e.,* class and methods) and discard the uncompilable parts. The Eclipse IDE provides a framework to apply similar parsing approaches to Java code: It identifies classes, methods and fields in a source file even though the compilation fails. We employed this framework to parse the code with the single constraint of being Java-dependent for this feature.

Since we do not have complete AST information for the code entities identified, code entities are treated as text and analyzed as natural language. The target entity is defined by the cursor position in the text editor, the nearest entity is picked as target entity.

When an entity is selected, the query is built by merging the obtained keywords obtained in two ways:

1) *Processing the entity's body.* We apply basic information retrieval techniques to extract the ten most frequent keywords in the body. We tokenize the entity's body on white spaces. For every token obtained, we split it on case change, digits and symbols. We lower the case and remove stop words. The set of tokens we obtain is ordered by frequency and the first ten become part of the query. To this set of keywords we add the entity's name. This is done because of Java interfaces. If the entity is a method, including the name would enhance the research. Being immutable, the method's name of a Java interface in a library, or a framework, is always the same. A Stack Overflow document would contain this method's name if one of the code snippets is tackling the implementation of a specific interface. For instance, a developer can invoke Seahawk on the method *decorate* implementing the *ILightweightDecorator* interface in the Eclipse API, an interface used to perform a custom decoration of the package explorer. In this situation, documents containing code snippets that implement the interface are enhanced because the term "decorate" is used. Moreover, the same term could be also used in the title of the document when questions regard the method.
2) *Analyzing the import statements.* We take all the import statements in the source file and remove the ones not used by the target entity. Since we do not have any information from the AST, 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 they reside in different packages, and are used by the same entity. However, we believe that such situations rarely happen. Once the imports are identified, we tokenize each statement on the "." character, and by defining a set of unique tokens that become part of the query. For example, assuming we have the following import statements used by an entity:

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

The resulting set of tokens would be [*java, util, List, ArrayList*].

The Annotation Engine

The recommendation engine allows the creation of links between source code and documents. To this aim, we implemented an annotation engine to let developers put annotations in the code. We want to allow developers to collaborate by means of the crowd knowledge itself. Differently from the query engine, which provides automated query generation, the annotation engine implements the second aspect of the manual interaction in the Seahawk recommendation system.

There are two main purposes in the annotation engine: creating and parsing annotations. The annotation structure must be flexible. To be language independent, we wanted to achieve this flexibility by embedding annotations in multi-line comments. Doxygen[9] follows a similar approach to integrate documentation in, for example, C++ and Java code and in Blueprint [19] 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.

Seahawk's approach gives users more flexibility. Developers can define custom delimiters (that need to match the target language' syntax for comments), and to avoid conflicts with Doxygen or JavaDoc annotations, we decided to put an exclamation mark as last character for the opening delimiter (*e.g.,* in Java the opening delimiter would become /*! instead of /* while in XML it would become <!–! instead of <!–). Listing 1 presents an example of Seahawk's annotation.

Listing 1: Example of Seahawk's annotation

```
/*!
* @documentId <Document's Id>
* @title <Document's title>
* @comment <Author's comment>
* @author <Autor's name>
* @creationTime <creation date>
*/
```

Whenever an annotation is created, Seahawk reports 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 identifies the target document to be suggested. The other fields are used to implement the basis of the support for collaboration. Seahawk does not explicitly provide collaborative functionalities, but relies on the fact that a versioning system (*e.g.,* Git, SVN) is used in the development phase. Putting annotations in the code is 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. Whenever a developer updates the repository, the new annotations are updated together with the comment explaining the purpose of the linked document. The role of the *comment*, *author* and *creationTime* fields guarantees that annotations are unique. The *comment* field is mainly used to allow developer to communicate with each other through the annotation system.

The annotation engine provides also a notification system to keep track of the annotations already seen by developers. For that reason we use two different ways of parsing code:

1) We implemented our own parser for annotations.
2) We took advantage of Eclipse's partitioning system.

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

The partitioning system identifies code blocks (partitions) that match specific delimiters in the code editor (*e.g.,* comment, classes, methods *etc.*). We ask it to match the annotation's delimiter and notify in case of changes. Whenever a source file is opened or modified in the code editor, the partitioning system notifies the view showing the suggested documents and storing the annotations in the cache, thus tracking the annotations that the developers have already seen. The latter relies on our implementation of the parser that works in the background. Whenever a project is updated, it parses all the updated files of the project and extracts annotations. If the annotations are not present in the annotation cache, they are considered as new annotations to be notified to the developer.

D. The User Interface

In Section II we presented an overview of the user interface of SEAHAWK. In this section we present each UI element and the functionalities provided to developers.

Document Navigator View

The first view of SEAHAWK is the one implementing the manual interaction of the recommendation engine. Through this view, developers can compose a query, send it to the search engine, and retrieve Stack Overflow's documents. Users are provided with a tree navigation system that allow them to explore single nodes (*i.e.,* questions or answers) of a single discussion. We reach the granularity of the code snippets in case they are available. By means of *drag and drop* (D&D) interactions, developers can drag a document or a code snippet into the code editor. Whenever a document is dropped in the editor, SEAHAWK shows a dialog (see Figure 3) where the user can put a comment to explain the link between the document and the code, and then it generates the annotation in the code editor.

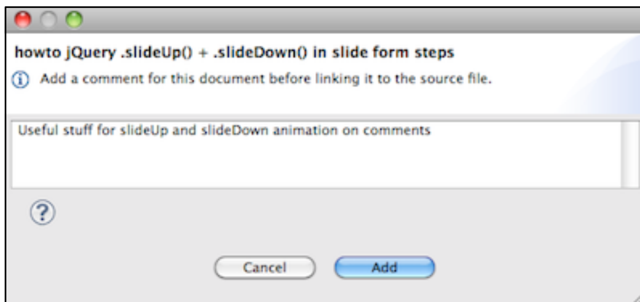

Figure 3: SEAHAWK dialog for annotation's comment

Suggested Documents View

Figure 1 (2) depicts a tree view similar to the one previously presented (1). which is used by SEAHAWK to show document linked to the code. Instead of presenting documents retrieved from a query, this view tightly works with the annotation engine. Whenever an editor become active, the annotation engine parses the file, extracts all of SEAHAWK's annotations, and notifies the view. The set of documents linked by the annotations is then retrieved from the search engine and displayed. Differently from the document navigator view, the user cannot drag documents in the code editor to create annotations. Allowing this feature would create redundancies in the annotations for documents that are already present in the code editor. Through a contextual menu, users can modify the comment of an annotation or delete the annotation as well. Annotation data is accessible by a tool tip that appears on top of the document when the mouse pointer is over it.

Since there is no mechanism to ensure consistency in the annotations, a linked document could have been removed in the search engine. In this situation, the document is shown anyway but the message "[Not Available]" is put in front of the document's title and it becomes not traversable (see Figure 4).

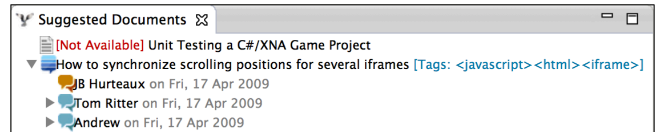

Figure 4: Document not available in SEAHAWK's view

Document's Contents View

When a document, or a document's node, is selected in one of the other views, the content is displayed in this view. To display the content, we use a custom layout and we take advantage of a web-browser widget embedded in the view. The web-browser widget allows the developer to navigate the links contained in the document and getting additional information. We use a Javascript library[10] to highlight the syntax of the text contained in the <*code*> tags, without having to care about the programming language. Questions are orange, the accepted answer is green and the other answers are blue.

Notification System

To rapidly spot new annotations in the project, we implemented notification system in the package explorer (Figure 1 (4)). Whenever a project is refreshed, the annotation engine parses the files and creates a list of annotations. Subsequently, it counts the number of annotations not seen and decorates the package explorer with the number of new annotations between square brackets. 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.

[10]http://code.google.com/p/google-code-prettify/

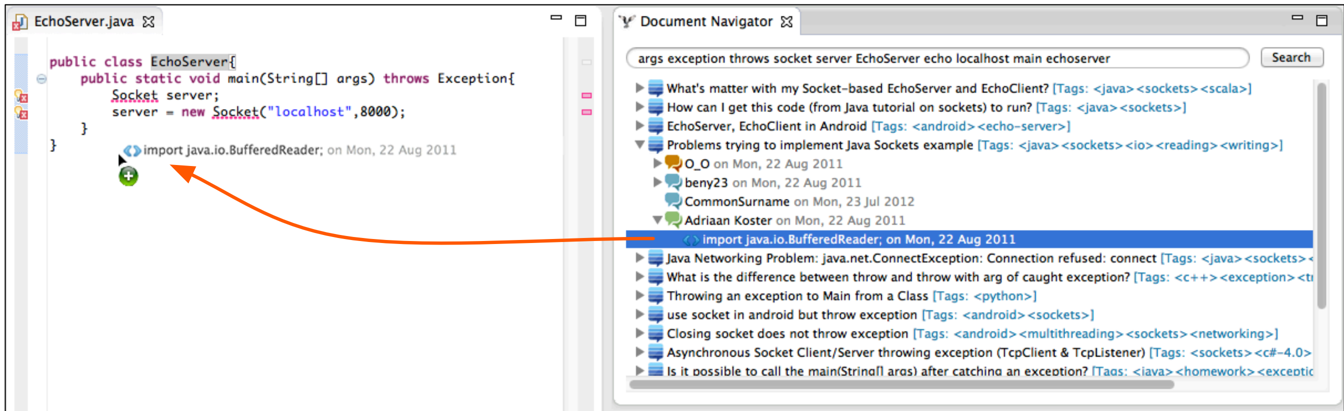

Figure 5: Alice imports the code snippet in the code editor.

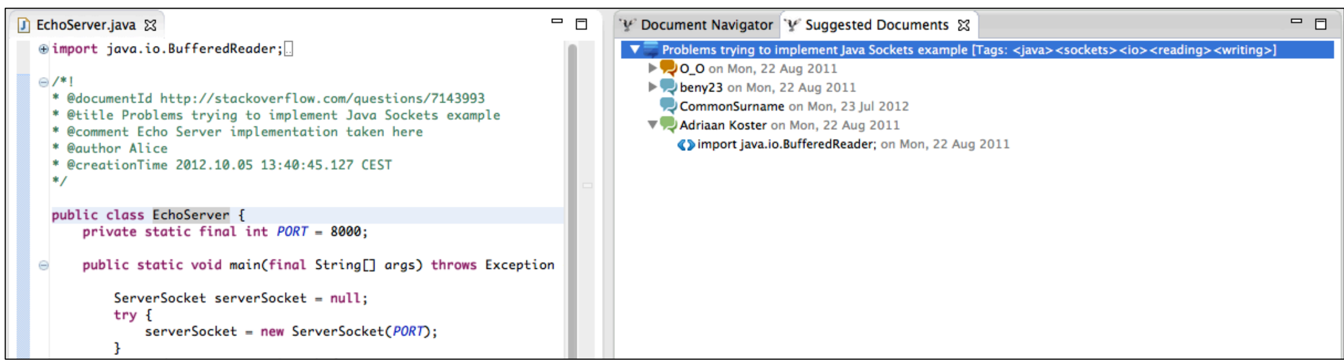

Figure 6: Notification of the linked document in the Suggested Documents View.

III. A Use Case Scenario

By means of a simple scenario, we illustrate how Seahawk can help developers to solve programming problems by leveraging Stack Overflowfrom within the Eclipse IDE.

Alice is required to build a simple *echo server* in Java. The echo server must handle one client at a time and it must terminate itself whenever a client sends the "quit" string.

Alice opens up the Eclipse IDE, with the Seahawk plugin installed, and begins creating the class *EchoServer*. She starts by creating a socket by using the *Socket* class:

Listing 2: Initial Implementation of an Echo Server

```
public class EchoServer{
    public static void main(String[] args){
        Socket server;
        server = new Socket("localhost",8000);
    }
}
```

Alice starts looking at the methods trying to find out a way to accept incoming connections. Since she does not find any method to accomplish this task, she invokes Seahawk through the contextual menu in the code editor.

Seahawk analyzes the existing code, builds a query, and retrieves a set of documents related to what is written in the *EchoServer* class (Figure 5). Among the documents, Alice finds out a question whose title is "Problems trying to implement Java Sockets". She reads the document and finds an accepted answer that proposes the implementation of a simple echo server. She realizes that the right class to be used instead of *Socket* is *ServerSocket*. Thanks to the document navigation system of Seahawk, she locates the code snippet and drags it into the code editor, importing it (Figure 5). Subsequently, Alice can start modifying the code in the editor to achieve the desired outcome. With minor modifications she adapts the imported snippet and makes the server able to be terminated when receiving a *quit* string from a connected client. In the end, Alice wants to bookmark the original solution directly in the code. Thus, she drags the document in the editor. Seahawk creates an annotation to link this specific Stack Overflow document and asks her to put a comment by means of a dialog box. Alice types the comment and confirms the creation of the annotation that becomes visible in the code editor. By doing so, every other person opening the file with the Seahawk plugin installed will be notified about the bookmark in an ad-hoc view (Figure 6).

IV. Evaluation

The previous scenario, while being a real example of Seahawk in action, does not provide any evidence in terms of usefulness and usability. To address the question whether Seahawk can actually help developers in their tasks, we present an evaluation composed of three different experiments.

A. Experiment I: Java Programming Exercises

We want to assess to what extent Seahawk can deal with plain text. We use a set of exercises taken from Java programming courses[11] to evaluate the relevance of the documents retrieved from Stack Overflow by extracting keywords from the text of the exercises.

Seahawk is not designed to directly deal with plain text taken from exercises. We thus had to recreate the right conditions to allow Seahawk to extract keywords from the text of the exercises: Since it needs at least a Java entity, we manually create a class stub with a name that summarizes the topic, and put the entire text of the exercise as a comment before, or inside, the class body, as depicted in Listing 3.

Listing 3: Example of Java exercise prepared for the test

```
/* Write a class that implements the CharSequence inter-
face 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 { }
```

With this approach 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 decided to use such a threshold because it is the smallest number of documents retrieved by Stack Overflow. We manually inspected and evaluated every document. With the term "relevant", we mean that the discussion can lead to a solution of the exercise either through the discussed topic or the code snippets. For example, the exercise in Listing 3 could lead to discussions tackling the implementation of *CharSequence* interface that could be partially relevant as well. For this reason, a binary notion of relevance is not enough. Thus, we defined *five* levels of relevance, ranging from 0 to 4. To have a numerical assessment of this experiment, we refer to the *normalized discounted cumulative gain* (NDCG), which is generally used to evaluate ranked retrieval results from search engines, using a multi-valued notion of relevance [17]:

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

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

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 is equal to 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 $Z_{kj} \sim 0.011$.

Experiment I: Results

The result we obtained from the NDCG index is 9.07%, thus meaning that one in ten of the documents retrieved was relevant to the Java exercises we used. In Figure 7 we present only a subset of the results, the ones we discuss afterwards; for the results for all the 35 exercises we refer to [15].

Exercise	D 1	D 2	D 3	D 4	D 5	D 6	D 7	D 8	D 9	D 10	D 11	D 12	D 13	D 14	D 15
Electrical Resistance	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Fibonacci	2	3	3	0	0	0	2	3	3	3	3	3	3	3	4
Metropolis	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Natural Merge Sort	3	3	4	0	3	0	4	3	0	0	0	3	2	2	2
Roulette Strategy	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Sudoku Solver	3	4	3	2	0	0	0	0	0	0	0	0	0	0	1
Wind Speed	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 7: Experiment I Results *(0 = Not Relevant, 4 = Highly Relevant)*

Although the *NDCG* value is low, there are some considerations to make. Our approach fails on very simple exercises: 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) provide very little information, thus the documents returned were unrelated. Sometimes the topic of the exercise was a subset of a more complex one. For instance, *RouletteStrategy* requires to calculate the number of turns required to lose all by betting only on red or black at roulette. The retrieved documents were discussing the same topic but at a higher level of difficulty (*e.g.,* machine learning approach), making them not relevant.

A reason to justify the irrelevance of the documents could reside in the absence of information in the Stack Overflow's crowd knowledge. Even though Stack Overflow archives many discussions on homework, the requirements of the exercises were not specific enough. Just in one case the exercise was in one of the document returned. Moreover, exercises requiring the implementation of data-classes (*e.g., Metropolis*) returned unrelated documents.

However, when an exercise tackles a well known topic (*i.e., Fibonacci*, *NaturalMergeSort* and *SudokuSolver*), the relevance of the documents increases: We were able to find solutions or even the full implementations in pseudocode, Java or similar languages that could be easily adapted and used to solve the exercise.

B. Experiment II: Method Stubs

In this experiment we wanted to assess the impact of Seahawk when dealing with method stubs. The scenario concerns a developer who starts to implement a method, does not know how to proceed, and asks Seahawk for help.

We selected eight different methods from student projects, and two exercises taken from a Java programming course, reaching a total of ten methods. Half of the methods were implementing part of a Java interface, the remaining half were regular methods. In doing so, we want to see if the behavior, in case of interfaces, changes with respect to regular methods. We removed the bodies from the methods to obtain stubs, leaving everything else unchanged.

Experiment II: Results

In Figure 8 we report the results for the stubs we tested.

Method Type	Class(method)	D 1	D 2	D 3	D 4	D 5	D 6	D 7	D 8	D 9	D 10	D 11	D 12	D 13	D 14	D 15
Interface	EnumerationImpl (hasMoreElements)	0	0	3	4	0	0	4	0	0	0	0	0	0	0	0
Interface	IntegerList (addAll)	2	2	2	3	1	0	2	0	0	1	0	0	1	2	2
Interface	MarkerInitActionDelegate (selectionChanged)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Interface	PreferencePaneMbox (createFieldEditors)	0	1	0	0	0	2	0	0	0	0	0	0	0	0	0
Interface	REmailLightweightDecorator (decorate)	4	2	4	0	0	0	0	0	0	0	0	0	0	0	0
Regular	CopyPaste (copy)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Regular	MarkerInitActionDelegate (prepareSQLite)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Regular	Parser (parseFunction)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Regular	SpreadsheetReader (loadFile)	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1
Regular	SpreadsheetReader (removeDoubleQuotes)	0	2	1	1	0	1	0	0	0	1	0	0	0	0	0

Figure 8: Experiment II Results *(0 = Not Relevant, 4 = Highly Relevant)*

When dealing with method stubs, Seahawk performs well with interface methods, but not with regular methods. Our conjecture is on the one hand that interface methods are less volatile than general methods, and on the other hand there is a higher probability that they have been discussed on Stack Overflow because interfaces are used by potentially many clients. For instance, for the *REmailLightweightDecorator (decorate)* method, Seahawk retrieves useful documents with the right code examples to implement a fully working decorator. Regular methods, on the other hand, have a lower probability to be discussed on Stack Overflow because they pertain to the specific domain of a system. The exception (see the results for the last two stubs) is when the "theme" of a method signature is of general interest (*e.g.,* loading a file, removing quotes). In the case of *SpreadsheetReader (removeDoubleQuotes)* the retrieved documents lead to a better solution than the one implemented in the original method. This is an argument in favor of having appropriate names for methods (*i.e.,* if removeDoubleQuotes would have been called rDQ Seahawk would have performed poorly).

C. Experiment III: Method Bodies

In the third experiment we want to assess the behavior of Seahawk when dealing with fully implemented methods. The scenario pertains to program comprehension: A developer invokes Seahawk to understand an existing and fully implemented method.

We selected seven fully implemented methods, one of which was implementing an interface. We left all methods unchanged, including comments. We wanted to see if the documents retrieved by Seahawk would help a developer achieving the same goal of the original implementation, thus helping her in getting a better understanding of the code.

Experiment III: Results

In Figure 9 we report the results for the tested methods.

Method Type	Class(method)	D 1	D 2	D 3	D 4	D 5	D 6	D 7	D 8	D 9	D 10	D 11	D 12	D 13	D 14	D 15
Interface	REmailLightweightDecorator (decorate)	0	0	0	0	0	0	0	0	3	0	0	0	0	0	0
Regular	MapEditor(buildMenu)	1	0	4	4	4	4	0	0	0	0	0	0	1	4	1
Regular	MapEditor(buildWest)	2	0	2	0	2	2	1	3	3	2	0	0	0	2	0
Regular	MarkerInitActionDelegate (prepareSQLite)	2	3	0	2	0	2	0	0	0	0	0	0	0	0	0
Regular	Parser (parseFunction)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Regular	SpreadsheetReader (loadFile)	2	1	2	0	0	0	4	4	2	2	0	0	0	4	0
Regular	SpreadsheetReader (removeDoubleQuotes)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 9: Experiment III Results *(0 = Not Relevant, 4 = Highly Relevant)*

Seahawk performs well in this scenario. On the one hand this is influenced by the higher amount of useful information that can be fed to the query engine. On the other hand the fact that methods have (or should have) a single responsibility limits the scope in a positive way, leading to more useful retrieved documents. Moreover, the library or framework used in the implementation of a method can refine the scope of the research. For example, many useful documents were retrieved for the methods *buildMenu* and *buildWest* of the class *MapEditor*. In this case, both methods largely used the *Swing* framework, restricting the search scope. The contrary happens when the topic of a method is too general. Moreover, as in the case of *parseFunction* of the class *Parser*, Seahawk is not able to retrieve relevant documents when the implementation of a method totally relies on code belonging to the project (*i.e.,* no particular standard libraries or well-known frameworks are used). A last scenario in which Seahawk performs poorly is when it is confronted to badly implemented methods. Tying this back to the program comprehension scenario: Sometimes understanding a method is not easy because the method is badly implemented. This also applies for cases where the signature of a method does not correspond to its implementation.

D. Reflections

The goal of our experiments was to assess the potential of SEAHAWK. While it may not always provide useful documents, SEAHAWK is capable of sometimes coming up with surprising insights that aid a developer both for program comprehension and software development. Moreover, time plays in favor of tools like SEAHAWK: the knowledge base of sites like Stack Overflow is constantly increasing. This on the positive side means that there are more and more relevant documents in Stack Overflow that can aid developers, on the negative side it means that dealing with information overload and noise reduction will become future issues.

V. RELATED WORK

Treude *et al.* [11] investigated the Stack Overflow service by analyzing randomly sampled data of the November 2010 data dump. They claim that Stack Overflow is particularly effective for code reviews, for conceptual questions, and for novices. In their subsequent work, they discussed the impact of web content curated by the crowd on software developers and their working practices [20]. They posed questions regarding the impact of social media on programming knowledge about software engineering education, and how it could influence the attitude of programmers. Storey *et al.* [21] 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.

Our work also lies in the field of search engines and code sample retrieval from the Web. Umarji *et al.* [22] investigated developers' habits in searching code on the internet. Sim *et al.* [23] investigated how sites for general purpose information retrieval (*e.g.,* Google) outperform custom sites for code search (*e.g.,* Krugle) and component reuse (*e.g.,* SourceForge) in retrieving code samples from the internet. In our approach we perform code retrieval on Stack Overflow. When samples are retrieved from search engines, the developer has to assess their validity. Since we rely on Stack Overflow, the code samples are already assessed by the community.

Laugher and Rodden [24] integrated in-project knowledge with an annotation system to link documentation and discussions regarding design decisions at source code level. Our approach follows a similar idea but does not force the user to look first at the code. SEAHAWK provides, through the *suggested documents view*, all the documents linked to the current editor even if annotations are not in focus.

Sawadsky *et al.* presented FISHTAIL [25], an Eclipse plugin built on top of MYLYN[12] [26]). FISHTAIL automatically suggests code examples from the web according to the most changed program element's name. In our approach, we do not only focus on the entity's name but we also use keywords to restrict the scope of the research.

[12]http://www.eclipse.org/mylyn/

Holmes *et al.* presented DEEPINTELLISENCE [27], a plugin for the Visual Studio IDE[13] that links bug reports, emails, events history and people to source code entities. Similarly, Čubraniç *et al.* created HIPIKAT [28], a recommender system to assist newcomers by recommending items from problem reports, newsgroup, and articles. Instead of providing resources from in-project knowledge, we focus on documents and discussions that are not related to the project.

Brandt *et al.* presented BLUEPRINT [19], a plugin built on top of Adobe Flex Builder that allows developers to search and import code examples in the IDE. Similarly, Zagalsky *et al.* presented EXAMPLE OVERFLOW [29] a web-based tool to search and recommend Javascript code samples. In SEAHAWK, we give the freedom of importing code samples of any kind and to link documents to any language. Developers can define their own annotations style to fit the desired target language.

Goldman *et al.* presented CODETRAIL [30] a plugin that connects source code to web resources by synchronizing Firefox[14] activities with Eclipse activities. In SEAHAWK we embed the web resources in the IDE to give more freedom to the user interactions (*e.g.,* code import).

Cordeiro *et al.* [31] presented an Eclipse plugin to help developers in problem solving tasks. Based on an exception's stack trace gathered from the IDE's console, they suggest related document from Stack Overflow. Instead of focusing on stack traces, we focus on the code written by the developer to suggest documents and we provide keywords as a query to retrieve documents.

Takuya *et al.* presented SELENE [32], an Eclipse plugin to spontaneously suggest code snippets to the developer. In our approach, importing code snippets is a side effect. We suggest entire discussions taken form Stack Overflow to enrich the information provided by code snippets.

VI. CONCLUSIONS

We presented a novel approach to leverage the Q&A crowd knowledge. We presented the implementation of our approach, SEAHAWK. SEAHAWK lets users interact with Stack Overflow documents in a novel way to import code snippets and create links between documents and source code by means of language-independent annotations, and how developers can use annotations to take advantage of the versioning system to collaborate and suggest documents to teammates. We also presented an approach to automatically generate queries from code entities, and we discussed how SEAHAWK deals with import statements and uncompilable code to extract keywords from Java code entities. Finally, we presented an evaluation of SEAHAWK and a discussion of the promising results we obtained.

Acknowledgements. We thank the Swiss National Science foundation for the financial support through SNF Project "SOSYA", No. 132175.

[13]http://www.microsoft.com/visualstudio/en-us

[14]http://www.mozilla.org/

References

[1] M. Robillard, R. Walker, and T. Zimmermann, "Recommendation systems for software engineering," *IEEE Software*, pp. 80–86, 2010.

[2] B. Lientz and B. Swanson, "Problems in application software maintenance." *Communications of ACM*, no. 11, pp. 763–769, 1981.

[3] T. Corbi, "Program understanding: Challenge for the 1990s," *IBM Systems Journal ()*, pp. 294–306, 1989.

[4] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining mental models: a study of developer work habits," in *Proceedings of ICSE 2006 (28th ACM International Conference on Software Engineering)*. ACM, 2006, pp. 492–501.

[5] A. J. Ko, R. DeLine, and G. Venolia, "Information needs in collocated software development teams," in *Proceedings of ICSE 2007 (29th ACM/IEEE International Conference on Software Engineering)*. IEEE CS Press, 2007, pp. 344–353.

[6] M. Hertzum and A. M. Pejtersen, "The information-seeking practices of engineers: searching for documents as well as for people," *Information Processing and Management: an International Journal*, 2000.

[7] J. Brandt, P. Guo, J. Lewenstein, M. Dontcheva, and S. Klemmer, "Two studies of opportunistic programming: interleaving web foraging, learning, and writing code," in *Proceedings of CHI 2009 (27th international conference on Human factors in computing systems)*. ACM, 2009, pp. 1589–1598.

[8] L. A. Adamic, J. Zhang, E. Bakshy, and M. S. Ackerman, "Knowledge sharing and yahoo answers: everyone knows something," in *In Proceedings of WWW 2008 (17th international conference on World Wide Web)*. ACM, 2008.

[9] K. K. Nam, M. Ackerman, and L. Adamic, "Questions in, knowledge in?: a study of naver's question answering community," in *In Proceedings of CHI 2009 (27th international conference on Human factors in computing systems)*. ACM, 2009, pp. 779–788.

[10] L. Mamykina, B. Manoim, M. Mittal, G. Hripcsak, and B. Hartmann, "Design lessons from the fastest q&a site in the west," pp. 2857–2866, 2011.

[11] C. Treude, O. Barzilay, and M.-A. Storey, "How do programmers ask and answer questions on the web? (nier track)," in *Proceedings of ICSE 2011 (33rd International Conference on Software Engineering)*. ACM, 2011, pp. 804–807.

[12] J. Raskin, *The Humane Interface - New Directions for Designing Interactive Systems*. Addison-Wesley, 2000.

[13] B. Dagenais and M. Robillard, "Recommending adaptive changes for framework evolution," in *In proceedings of ICSE 2008 (30th international conference on Software engineering)*. ACM, 2008, pp. 481–490.

[14] A. Bacchelli, L. Ponzanelli, and M. Lanza, "Harnessing stack overflow for the ide," in *In Proceedings of RSSE 2012 (3rd International Workshop on Recommendation Systems for Software Engineering)*, 2012, pp. 26–30.

[15] L. Ponzanelli, "Exploiting crowd knowledge in the ide," Master's thesis, University of Lugano, 2012.

[16] M. McCandless, E. Hatcher, and O. Gospodnetic, *Lucene in Action, Second Edition: Covers Apache Lucene 3.0*. Manning Publications Co., 2010.

[17] C. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[18] L. Moonen, "Generating robust parsers using island grammars," in *Proceedings of WCRE 2001 (8th Working Conference on Reverse Engineering)*. IEEE CS, 2001, pp. 13–22.

[19] J. Brandt, M. Dontcheva, M. Weskamp, and S. R. Klemmer, "Example-centric programming: integrating web search into the development environment," in *In Proceedings of CHI 2010 (28th international conference on Human factors in computing systems)*. ACM, 2010, pp. 513–522.

[20] C. Treude, "Programming in a socially networked world: the evolution of the social programmer," pp. 1–3, 2012.

[21] M.-A. Storey, C. Treude, A. van Deursen, and L.-T. Cheng, "The impact of social media on software engineering practices and tools," in *In Proceedings of FoSER 2010 (FSE/SDP workshop on Future of software engineering research)*. ACM, 2010, pp. 359–364.

[22] M. Umarji, S. Sim, and C. Lopes, "Archetypal internet-scale source code searching," in *In Proceedings of OSS 2008 (4th International Conference on Open Source Systems)*, 2008, pp. 257–263.

[23] S. Sim, M. Umarji, S. Ratanotayanon, and C. Lopes, "How well do search engines support code retrieval on the web?" *ACM Trans. Software Engineering Methodologies*, pp. 1–25, 2011.

[24] R. Lougher and T. Rodden, "Supporting long-term collaboration in software maintenance," in *In Proceedings COCS 1993 (Proceedings of the conference on Organizational computing systems)*. ACM Press, 1993, pp. 228–238.

[25] N. Sawadsky and G. Murphy, "Fishtail: from task context to source code examples," in *Proceedings of TOPI 2011 (1st Workshop on Developing Tools as Plug-ins)*. ACM, 2011, pp. 48–51.

[26] M. Kersten and G. Murphy, "Using task context to improve programmer productivity," in *Proceedings of SIGSOFT 2006/FSE-14 (14th ACM SIGSOFT international symposium on Foundations of software engineering)*. ACM Press, 2006, pp. 1–11.

[27] R. Holmes and A. Begel, "Deep intellisense: a tool for rehydrating evaporated information," in *In Proceedings of MSR 2008 (5th international working conference on Mining software repositories)*. ACM, 2008, pp. 23–26.

[28] D. ČubraniĆ, G. MurphyC, J. Singer, and K. Booth, "Learning from project history: a case study for software development," in *In Proceedings of CSCW 2004 (ACM conference on Computer supported cooperative work)*. ACM, 2004, pp. 82–91.

[29] A. Zagalsky, O. Barzilay, and A. Yehudai, "Example overflow: Using social media for code recommendation," 2012.

[30] M. Goldman and R. Miller, "Codetrail: Connecting source code and web resources," *Journal of Visual Languages & Computing*, pp. 223–235, 2009.

[31] J. Cordeiro, B. Antunes, and P. Gomes, "Context-based recommendation to support problem solving in software development," in *In Proceedings of RSSE 2012 (3rd International Workshop on Recommendation Systems for Software Engineering)*, 2012.

[32] W. Takuya and H. Masuhara, "A spontaneous code recommendation tool based on associative search," in *Proceedings of SUITE 2011(3rd International Workshop on Search-Driven Development: Users, Infrastructure, Tools, and Evaluation)*. ACM, 2011, pp. 17–20.