Mining StackOverflow to Turn the IDE into a Self-Confident Programming Prompter

Luca Ponzanelli[1], Gabriele Bavota[2], Massimiliano Di Penta[2], Rocco Oliveto[3], Michele Lanza[1]
1: REVEAL @ Faculty of Informatics – University of Lugano, Switzerland
2: University of Sannio, Benevento, Italy 3: University of Molise, Pesche (IS), Italy

ABSTRACT

Developers often require knowledge beyond the one they possess, which often boils down to consulting sources of information like Application Programming Interfaces (API) documentation, forums, Q&A websites, *etc.* Knowing what to search for and how is non-trivial, and developers spend time and energy to formulate their problems as queries and to peruse and process the results.

We propose a novel approach that, given a context in the IDE, automatically retrieves pertinent discussions from Stack Overflow, evaluates their relevance, and, if a given confidence threshold is surpassed, notifies the developer about the available help. We have implemented our approach in Prompter, an Eclipse plug-in. Prompter has been evaluated through two studies. The first was aimed at evaluating the devised ranking model, while the second was conducted to evaluate the usefulness of Prompter.

Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Programming Environments—*Interactive environments*

General Terms

Documentation, Experimentation

Keywords

Recommender Systems, Developers Support, Empirical Studies

1. INTRODUCTION

The myth of the lonely programmer is still lingering, in stark contrast with reality: *Software development, also due to the ever increasing complexity of modern systems, is tackled by collaborating teams of people*. A helping hand is often required, either by team mates [18], through pair programming sessions [6], or the perusal of vast amounts of knowledge available on the Internet [41].

While asking team mates is the preferred means to obtain help [20], their availability may fall short. In this case, developers resort to electronically available information. This comes with a number of problems, the main one being the absence of automation: Every time developers need to look for information, they interrupt their work flow, leave the IDE, and use a Web browser to perform and refine searches, and assess the results. Finally, they transfer the obtained knowledge to the problem context in the IDE. The information is retrieved from different sources, such as forums, mailing lists [2], blogs, Q&A websites, bug trackers [1], *etc.* A prominent example is Stack Overflow, popular among developers as a venue for sharing programming knowledge. Stack Overflow is vast: In 2010 it already had 300k users, and millions of questions, answers, and comments [23]. This makes finding the right piece of information cumbersome and challenging.

Recommender systems [33] represent a possible solution to this problem. A recommender system gathers and analyzes data, identifies useful artifacts, and suggests them to the developer. Seminal tools, such as eRose [43], Hipikat [9] and DeepIntellisense [14], suggest project artifacts in the IDE aiming at providing developers with additional information on specific parts of the system. They come however with a caveat: the developer must proactively invoke them, and, once invoked, they continuously display information, which may defeat their purpose, as they augment the complexity of what is displayed in the IDE. *Ideally, a recommender system should behave like a prompter in a theatre: Ready to provide suggestions whenever the actor needs them, and ready to autonomously give suggestions if it feels something is going wrong*.

The interaction between the theatre prompter and the actor is similar to the interaction between two developers doing pair programming, working side by side to write code. These developers have different roles, *i.e.,* the driver, who is in charge of writing code, and the observer, who observes the work of the driver [42], tries to understand the context, and, if she has enough confidence, interrupts the driver by giving suggestions. In addition, the driver can consult the observer whenever she needs it, making the observer the programming prompter of the programming actor.

This interaction is what we propose in Prompter, a tool that automatically retrieves and recommends, with push notifications, relevant Stack Overflow discussions to the developer. Prompter makes the IDE a programming prompter that silently observes and analyzes the code context in the IDE, automatically searches for Stack Overflow discussions on the Web, evaluates their relevance by taking into consideration *code aspects* (*e.g.,* code clones, type matching), *conceptual aspects* (*e.g.,* textual similarity), and Stack Overflow *community aspects* (*e.g.,* user reputation) to decide, given a certain amount of self-confidence (encoded in a threshold the user can change through a slider, to make the recommender quiet or talkative) when to suggest discussions. We have evaluated Prompter through two studies, one aimed at evaluating its ranking model, the other aimed at evaluating its usefulness to developers.

MSR '14, May 31 – June 1, 2014, Hyderabad, India

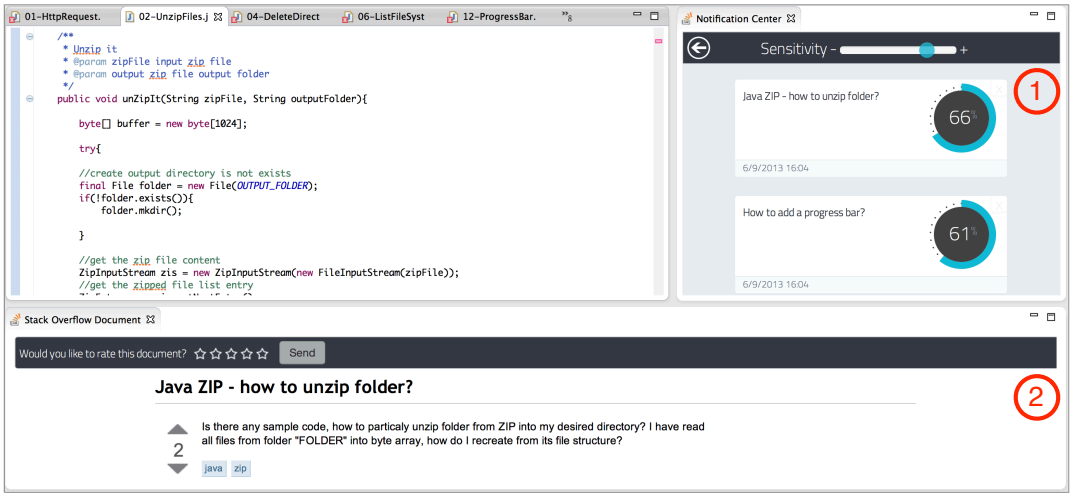

Figure 1: The PROMPTER User Interface.

Contributions. We make the following contributions:
1. a novel ranking model that evaluates the relevance of a Stack Overflow discussion, given a code context in the IDE, by considering code, conceptual and community aspects;
2. PROMPTER , an Eclipse plugin that implementats our approach;
3. an empirical evaluation, *Study I*, aimed at validating the devised ranking model;
4. a controlled experiment, *Study II*, aimed at evaluating the usefulness of PROMPTER during development and maintenance tasks;
5. a publicly available replication package for both studies[1].

2. APPROACH

We first detail the user interface and architecture of PROMPTER, the recommender system that implements our approach. We then describe the models and techniques that enable its self-confidence.

2.1 Prompter

User Interface. Figure 1 shows the user interface of PROMPTER. It provides two views through which the user can (i) receive and track notifications, and (ii) read the suggested Stack Overflow discussions.

The notification center (1) keeps track of the last ten notifications made by PROMPTER, reporting the title of the Stack Overflow discussions, the timestamp of the notification, and a percentage indicating PROMPTER's confidence with which it deems a Stack Overflow discussion relevant to the given IDE context. Whenever PROMPTER considers a discussion as relevant for the current context, it opens the notification center and plays a sound. If a Stack Overflow discussion is notified more than once, it is pushed to the top of the list for visibility. At the top of the notification center, the developer can change PROMPTER's sensitivity: by sliding to the right PROMPTER is more talkative, by sliding to the left it becomes more taciturn.

Whenever a developer clicks on a notification, a Stack Overflow document view (2) is opened, which shows the contents of the Stack Overflow discussion. It provides a rating bar at the top of the view, with which a developer can rate the current discussion. PROMPTER keeps track of the ratings and the evaluated documents. The rating bar does not appear in case a developer has already evaluated a document within the same code context. Like in pair programming, the observer may be silent, and the code writer may ask her companion for advice. We implemented the same interaction. The developer can manually invoke PROMPTER thorough a contextual menu in the code editor and the Eclipse package explorer.

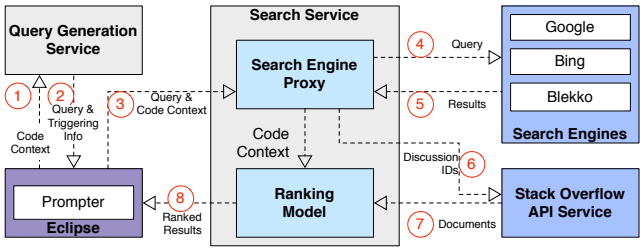

Figure 2: PROMPTER architecture.

Architecture. Figure 2 depicts the architecture of PROMPTER, which is composed of (i) the *Eclipse plug-in*, (ii) the *Search Service*, and (iii) the *Query Generation Service*. The numbers in the picture represent the sequence of actions that PROMPTER performs when it retrieves documents from Stack Overflow.

The Eclipse plug-in tracks code contexts (1) every time a change in the source code occurs. The extracted code context—code elements to formulate the query—is sent to the *Query Generation Service*, which formulates a query starting from the code context (2). It extracts a query and, according to a set of parameters described later, determines if a new search can be triggered. This information is sent back, with the query and the context, to the plug-in. Since the query is the basis of every search triggered by PROMPTER, the plug-in also considers the query when deciding to trigger a new search. To avoid repeated identical searches, PROMPTER submits a new search only if the query differs from the last one. The query and code context are sent to the *Search Service* (3), which acts as a proxy between the plug-in, the search engines to which the query is sent, and the Stack Overflow API. The query is sent to search engines (Google, Bing, Blekko) to perform a Web search on the Stack Overflow website (4). All resulting URLs are collected and duplicates removed (5). Every URL that refers to a question from Stack Overflow must match the form `stackoverflow.com/questions/<id>/<title>`, otherwise it is discarded. The service uses the Stack Overflow question ID to retrieve the discussion via the Stack Overflow API (6). Every

[1] `http://prompter.inf.usi.ch/`

discussion is, given the code context, ranked (7) according to the Prompter ranking model (see Section 2.2). The ranked list of URLs is sent back to the plug-in where Prompter decides whether to fire a new notification in the IDE (8).

2.2 Retrieval Approach

We present an integrated approach capable of (i) connecting different aspects of the code written by developers to the information contained either in the text or in the code of Stack Overflow discussions, and (ii) taking into consideration information about the quality of the discussions that Stack Overflow has available (*e.g.,* user reputation and questions/answers score). Our previous work in this context [28] only used text similarity as a means to retrieve Stack Overflow discussions related to the actual code. This led to errors in the identification of relevant discussions for the source code in the IDE. The approach we present here relies on several combined aspects, and has proven to be more robust and less error-prone.

2.2.1 Tracking Code Contexts in the IDE

Prompter is meant to be a silent observer "looking" at what a developer writes, with the aim of suggesting relevant Stack Overflow discussions. Whenever the developer types, Prompter waits until the developer stops writing, identifies the current code element (*i.e.,* method or class) that has been modified, and extracts the current context, which consists of: (i) a fully qualified name identifying the code element, *i.e.,*`packageName.ClassName` for classes and `packageName.ClassName.methodSignature` for methods; (ii) the source code of the modified element (*i.e.,* class or method); (iii) the types of the used API, taking into account only types outside the analyzed Eclipse project (*i.e.,* declared in external libraries or in the JDK); and (iv) the names of methods invoked in the API, again considering only external libraries and JDK only. The extracted information (*i.e.,* the context) is sent to the *Query Generation Service* (see Figure 2) to generate a query.

2.2.2 Generating Queries From Code Context

Since we want to automate the triggering of searches for discussions on Stack Overflow, we have to devise a strategy to build a query describing the current code context in the IDE. A näive approach [28] is to treat the code as a bag of words by: (i) splitting identifiers and removing stop words; (ii) ranking the obtained terms according to their frequency; and (iii) selecting the top-n most frequent terms. However, using only the frequency value is not highly discriminating in selecting terms that appropriately describe the context. For example, words like *print*, *run*, or *exception*, even if very frequent in source code and not considered stop words in English, have a too general meaning in programming to discriminate the programming context. Our solution is to also consider the entropy [8] of a given term t in Stack Overflow, computed as:

$$E_t = \sum_{d \in D_t} p(d) \cdot \log_{\mu} p(d) \quad (1)$$

where D_t is the set of discussions in Stack Overflow containing the term t, μ is the number of discussions in Stack Overflow, and $p(d)$ represents the probability that the random variable (term) t is in the state (discussion) d. Such a probability is computed as the ratio between the number of occurrences of the term t in the discussion d over the total number of occurrences of the term t in all the discussions in Stack Overflow. The entropy has a value in the interval of $[0, 1]$. The higher the value, the lower the discriminating power of the term. We computed the entropy of all terms present in Stack Overflow discussions by using the data dump of June 2013[2]. This resulted in entropy information for 105,439 different terms. Terms like the ones previously mentioned exhibit very high levels of entropy (*e.g.,* for *run* the entropy was 0.75) compared to less frequent and more discriminative terms (*e.g.,* for *swt* the entropy was 0.25). Therefore, term entropy can be used to lower the prominence of frequent terms that do not sufficiently discriminate the context. The *Query Generation Service* ranks the terms in the context based on a term quality index (*TQI*):

$$TQI_t = \nu_t \cdot (1 - E_t) \quad (2)$$

where t is the term, ν_t is frequency in the context, and E_t is its entropy value measured as described before. Once the raking is complete, the *Query Generation Service* selects the top n terms to devise the query, plus the word *java*. The query can exceed n terms in case two or more terms exhibit the same *TQI* value.

The term entropy approach has one drawback. We observed that terms with a very low entropy (thus good candidates to be part of a query) may be terms containing typos (*e.g.,* for *override* the entropy was 0.63, and for *overide* it was 0.05). They are present in very few Stack Overflow discussions and thus have a low entropy. To overcome this problem, before selecting the n terms to create the query, we use the *Levenshtein distance* [21] to verify if in the context there are terms with a very high textual similarity. If we detect two terms (say t_i and t_j) having *Levenshtein distance* = 1, the term having the lower frequency in the context (say t_i) is discarded and considered as a likely typo, and its frequency is added to the frequency of t_j. If the two terms have the same frequency, we discard the term with the lower entropy as the likely typo.

2.3 Prompter Ranking Model

The goal of the ranking model is to rank the retrieved Stack Overflow discussions, and assign them a value that measures their relevance to the query. It relies on 8 different features that capture relations between Stack Overflow discussions and source code.

1. Textual Similarity: The similarity of the code in the IDE to the textual part of a Stack Overflow discussion without code samples. The goal is to assess the similarity between the topics of the code and the topics of the discussion. We use Apache Lucene to create the index and preprocess the contents, by removing English stop words and Java language keywords, by splitting compound identifiers/token based on case change and presence of digits, and by applying the Snowball stemming. Finally, we compute the cosine similarity among the *tf-idf* vectors[3, 24].

2. Code Similarity: The percentage of lines of code in the IDE that are cloned in the Stack Overflow discussion.

3. API Types Similarity: The percentage of API types used in the code that are also present in the Stack Overflow discussion. These are types that are not declared in the project, but in external libraries or in the JDK. The higher the usage of the same types in both discussions and code, the more the potential usefulness of the discussions. To identify the API types, we parse every code sample in the discussion with the Eclipse JDT parser. We are able to resolve types among different samples in the discussion as long as the fully qualified name (*e.g.,* imports) of the type is used in one them, or if the identified type is part of the standard JDK. In case of unresolved types, we match the identified simple name of the class with the simple name of the types used in the code.

4. API Methods Similarity: The percentage of API method invocations used in the code that are present in the Stack Overflow discussion. Higher values suggest a similarity in usage of the API. We use the Eclipse JDT parser, which identifies method invocations that respect the Java grammar even if the type is not resolved. Since we can only identify the name and number of parameters without any

[2] `http://www.clearbits.net/torrents/2141-jun-2013`

signature, we only consider the name of the invoked method, which helps matching overloaded methods.

5. Question Score: The quality of the score of the question in the Stack Overflow discussion. Since the score is not bounded, we normalize the value in the range [0,1] using a sigmoid function:

$$\sigma(x) = \frac{1}{1 + e^{(\bar{x}-x)}} \tag{3}$$

where x is the score and $\bar{x}$ is the average of the scores of all the questions in Stack Overflow according to the data dump of June 2013. This index indicates the quality of the question, according to the Stack Overflow community.

6. Accepted Answer Score: The quality of the score of the accepted answers in the Stack Overflow discussion. In case no accepted answer is present, the score is set to zero. The score is normalized like the question score, using the related average score. This index indicates the quality of the accepted answer, according to the Stack Overflow community.

7. User Reputation: The level of reputation of the person who posted the question. The value is normalized like the two previous features, using the related average value. Differently from the two previous indexes, this index evaluates the reliability of the person who asked the question on the Stack Overflow community.

8. Tags Similarity: The percentage of tags covered by keywords extracted from imports. Tags are split on number and symbols to remove versions (*e.g., apache-httpclient-4.x* becomes *apache httpclient*), and tokens are matched against the tokens obtained by splitting imports on dots. The idea is to identify the topics or libraries used in the discussions even if there is no code in the discussion.

2.3.1 Definition & Calibration of the Ranking Model

These 8 features are linearly combined to define the ranking model. Each feature is assigned a weight that defines the impact of this specific feature on the overall score:

$$S = \sum_{i=1}^{n} w_i \cdot f_i \qquad \text{having} \qquad \sum_{i=1}^{n} w_i = 1 \tag{4}$$

where $f_i \in [0,1]$ is a feature value and $w_i \in [0,1]$ is the assigned weight. The score S ranges in the interval [0,1]. The next step is to calibrate the weights of the Prompter features in equation 4.

We need a way to objectively measure the recommendation accuracy of a given Prompter configuration, a "gold standard" composed of code contexts each of which is linked to a set of "relevant" (useful to a developer working on a specific context) Stack Overflow discussions. With such a dataset, the recommendation accuracy of a specific Prompter configuration can be easily measured as the number of code contexts for which Prompter is able to retrieve a relevant Stack Overflow discussion in the first position. Since Prompter recommends only one specific document (the top ranked one), we only need to evaluate the accuracy for that document.

To identify the best configuration we used an exhaustive combinatorial search. We measured the performance of all configurations obtained varying each weight between 0 and 1 with step size 0.01 where the weights total 1, as defined in equation (4). Although time-consuming, this avoids that a possible sub-optimal calibration affects the study results. Such a calibration process might be highly biased by the choice of the dataset, *i.e.,* of the set of code contexts. We tried to mitigate this threat by maximizing the dataset diversity, and its representativeness of various programming problems developers could encounter: We collected a large set of problems encountered by Master's and Bachelor's students during laboratory and project activities conducted in the context of software engineering courses. For each problem, we asked the students to provide a description and the code they produced before requesting help from their teacher aiming at deriving a solution. Since senior developers could have problems of different nature than students, we also asked industrial developers to collect problems they encountered during their development activity and to provide us samples of code they produced just before asking for help or searching for solutions.

We collected 74 code contexts, 48 from academic contexts and 26 from industry. We randomly sampled half of them (37) for the calibration, and used the remaining 37 for the first evaluation of Prompter described in Section 3. For each of the 37 contexts used for the calibration, we browsed Stack Overflow with the aim of finding pertinent, helpful, discussions. More than one discussion could be identified in this phase. The set of relevant documents manually identified represents our "gold standard" to measure the suggestion accuracy of a specific Prompter's configuration.

Table 1: Prompter Ranking Model: Best Configuration.

Index	Weight	Index	Weight
Textual Similarity	0.32	Question Score	0.07
Code Similarity	0.00	Accepted Answer Score	0.00
API Types Similarity	0.00	User Reputation	0.13
API Methods Similarity	0.30	Tags Similarity	0.18

Table 1 reports the configuration that provides the best recommendation accuracy. The indices with value 0.00 have been discarded from the model after completing the calibration. We have used this configuration for the two evaluation studies. Having 74 code contexts available (along with manually identified relevant documents), and having calibrated the model using only 37 of them, we could have used the other 37 contexts as a test set to automatically evaluate the performance of the ranking model. However, such an evaluation would have been biased by our manual validation of the links between contexts and relevant documents. Instead, we do not have such a threat in the studies we performed, because the relevance was evaluated by external participants (*Study I*), or with participants using Prompter in maintenance and development tasks (*Study II*).

2.4 Putting It Together

The result of the Prompter ranking model is not enough to determine if a discussion is to be recommended or not. As we discussed in Section 2.1, the user can define the sensitivity of Prompter in notifying new discussions, and we showed how the *Query Service* determines if a new search is to be triggered or not. Triggering a new search and notifying a discussion relies on two thresholds: (i) *Query Entropy Threshold* and (ii) *Minimum Confidence Threshold*. The former is sent to the *Query Service* and defines the entropy level that should not be exceeded by the median (or mean, depending on the user preferences) of the terms of the query. If the value is below the threshold, a new search is triggered. The latter defines the minimum confidence level needed for a discussion to be recommended. Both thresholds range in the interval $[0,1]$. We limited the interval to $[0.1, 0.9]$ to prevent Prompter from not being able to submit new searches or notify new discussions. Whenever one uses the sensitivity slider, these values are modified in an inverse proportional way. A complete slide to the right means a high-sensitive configuration with *Query Entropy Threshold* at 0.9 and *Minimum Confidence Threshold* at 0.1, and the opposite otherwise.

3. STUDY I: EVALUATING PROMPTER'S RECOMMENDATION ACCURACY

The *goal* of this study is to evaluate, from a developer's *perspective*, the relevance of the Stack Overflow discussions identified by Prompter, to understand to what extent the retrieved discussions

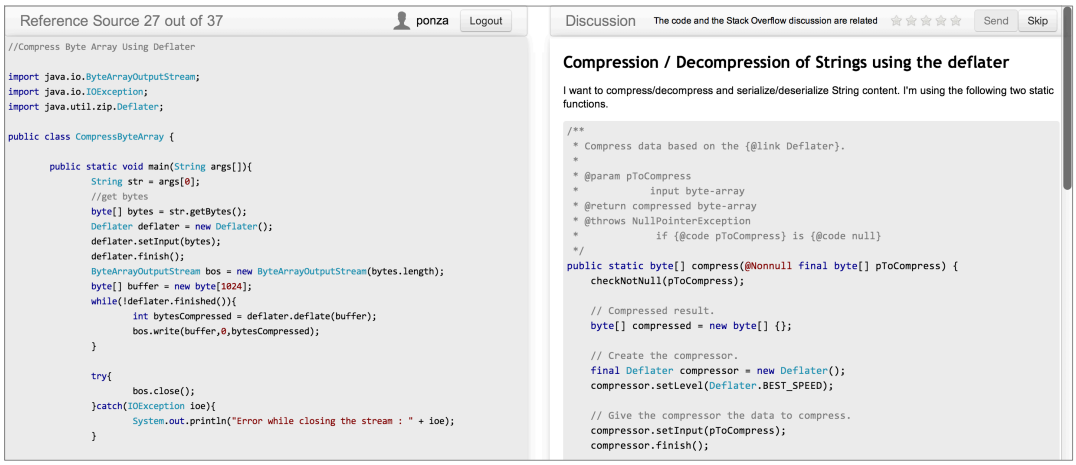

Figure 3: An Example Question from Our Questionnaire.

provide useful information to a developer working on a particular code snippet.

3.1 Study Design and Planning

The *context* of the study consists of *participants*, *i.e.,* various kinds of developers, among professionals and students, and *objects*, *i.e.,* source code snippets and its related Stack Overflow discussion as identified by PROMPTER. The study addresses the following research question (RQ_1): *To what extent are the SO discussions identified by* PROMPTER *relevant for developers?*

We aim at investigating to what extent the Stack Overflow discussions identified by PROMPTER contain information perceived as relevant by developers for a specific programming task. We asked 55 people (industrial developers, academics, and students) to complete a questionnaire aimed at evaluating the relevance of the Stack Overflow discussions identified by PROMPTER, by analyzing a specific code snippet. 33 participants filled in the questionnaire: 13 developers, 9 PhD students, 7 MSc students, 2 BSc students, 2 faculty. All participants had experience in software development and design. The least experienced participants were 3rd year BSc students that practiced software development and documentation production in the context of a Software Engineering course.

The different background of participants is a *requirement* for this study, since PROMPTER should be able to support developers having different skills, programming knowledge, and experience. Participants answered the questionnaire through a Web application. They received the URL of the questionnaire, along with email instructions. Before accessing the questionnaire, participants were required to create an account, and to fill in a pre-questionnaire aimed at gathering information on their background.

Once the participants answered the pre-questionnaire, they had to perform (up to) 37 tasks where the web application showed a Java class and a discussion from Stack Overflow that PROMPTER suggested as top-1 ranked discussion among the results retrieved when analyzing that class. Even though participants had the chance of skipping tasks, we obtained at least 30 answers for each task. In the context of this study, we used the remain 37 code snippets manually collected as explained in Section 2.3.1. Participants had to express their level of agreement to the claim *"The code and the Stack Overflow discussion are related"*, providing a score on a five points Likert scale [27]: 1 (strongly disagree), 2 (disagree), 3 (neutral), 4 (agree), and 5 (strongly agree). In other words, the participants had to indicate to what extent the discussion could help them in competing the implementation task in the showed class.

Figure 3 shows an example of task from our survey. After submitting the score, participants were asked to write an optional comment to explain the rationale of their evaluation. We gave participants four weeks to complete the questionnaire. The participants were neither aware of the experimented technique (*i.e.,*PROMPTER) nor how the Stack Overflow discussions were selected. The web questionnaire was also designed to (i) show the 37 tasks to participants in random order to limit learning and tiredness effects, and (ii) measure the time spent by each subject in answering each question. Response time was collected to remove from the analysis of the results participants that provided answers in a less than 10 seconds, *i.e.,* without carefully reading code and the Stack Overflow discussion. This was not the case for any participant.

3.2 Analysis of the Results

We quantitatively analyzed participants' answers through violin-plots [12] to assess the ability of PROMPTER in identifying relevant Stack Overflow discussions given a piece of code. Violin plots combine box-plots and kernel density functions to better indicate the shape of a distribution. The dot inside a violin plot represents the median. A thick line is drawn between the lower and upper quartiles, while a thin line is drawn between the lower and upper tails. We also e qualitatively analyzed the feedback provided by the participants as well as cases of discussions particularly appreciated/disliked by participants to identify strengths and weaknesses of PROMPTER.

Figure 4 shows the violin-plots of scores provided by participants of our experiment to each of the 37 questions composing our questionnaire (*i.e.,* their level of agreement to the claim *"the code and the Stack Overflow discussion are related"*). To understand whether PROMPTER excels in particular domain, we grouped the 37 tasks based on the topic/piece of technology they are related to, instead of ordering the tasks by their number. Overall, the analyzed Stack Overflow discussions have been considered related to the showed Java code snippet. 28 out of the 37 analyzed discussions (76%) received a median score greater than equal to 4. This means that participants agreed or strongly agreed to the above reported claim. Among the remaining 9 discussions, 5 (14%) achieved 3 as median, meaning that participants were generally undecided about their relevance to the code context, and four (10%) were mostly marked as not relevant achieving a median score of 2 (*i.e.,* disagree). In the following, we discuss two examples in which PROMPTER performed

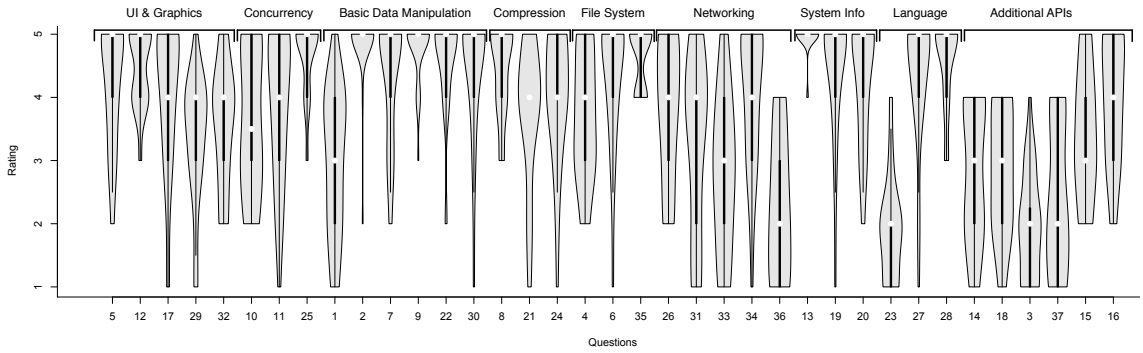

Figure 4: Violin Plots of Scores Assigned by Participants to the Evaluated Stack Overflow Discussions.

well, and a scenario in which we show its limitations.

Example I. The question reported in Figure 3 (question 8 in Figure 4) is an example where the achieved median score is 5. The class `CompressByteArray`—implementing the compression of a `byte` array using the `Deflater` class—has been linked by PROMPTER to the Stack Overflow discussion **Compression/Decompression of Strings using the deflater**[3]. Among the comments left by developers to their votes, one explained its "strongly agree" vote with the following sentence: *it is a good discussion if working on the* `CompressByteArray` *class, since it talks about compression with deflater, decompression, but also about problems that could be experienced and possible solutions*.

Example II. Another Stack Overflow discussion felt by developers as strongly related to the companion Java class was the one entitled **Java regex email**[4] (question 9 in Figure 4) and associated to the following `Utility` class:

```
import java.util.regex.*;

public class Utility {
    public static boolean isValidEmailAddress(String email) {
    //regex to match an e-mail address
        String EMAIL_REGEX = "^[\\w-_\\.+]*[\\w-_\\.]\@([\\w]+\\.)+[\\w]";
        Pattern emailPattern = Pattern.compile(EMAIL_REGEX);
        Matcher emailMatcher = emailPattern.matcher(email);
        return emailMatcher.matches();
    }
}
```

The `Utility` class emulates a developer experiencing troubles in writing the method `isValidEmailAddress`, aimed at validating through the Java `regex` mechanism an email address provided as parameter. In particular, the regular expression stored in variable `EMAIL_REGEX` is wrong, and for this reason `isValidEmailAddress` is incorrect. In the Stack Overflow discussion retrieved by PROMPTER as the most related one to the `Utility` class, a user is asking help since she is experiencing a similar problem when trying to validate an email address using Java `regex`. The top answer in this discussion contains the solution to the problem in method `isValidEmailAddress`, *i.e.,* the correct regular expression to validate email addresses. This explains why almost all subjects involved in our study (26 out of 32) assigned a score equal to 5 to this discussion.

Example III. Developers did not consider particularly useful the discussion **Invoke only a method of a servlet class not the whole servlet**[5] related to the `ShoppingCartViewerCookie` servlet class (question 36 in Figure 4). The reason why PROMPTER linked `ShoppingCartViewerCookie` to this discussion is because it is about servlets, but not about the particular problem the developer wants to solve (*i.e.,* managing cookies). Instead, the discussion explains how to invoke a single method of a servlet. This was also confirmed by one of the participants: *"the SO discussion does not mention how to use cookies"*. This example shows the limits of PROMPTER: It correctly captures the general context of the code (a developer is working on a servlet class), but it fails to identify the problem she is experiencing when trying to implement a specific feature. The same happened in the few cases where our approach obtained low scores (questions 3, 23, and 37 in Figure 4).

Summary of Study I. *The Stack Overflow discussions identified by* PROMPTER *are, from a developer's point-of-view, generally considered related to the source code. In particular, 76% of the discussions where considered related (median 4) or strongly related (median 5) by developers, while only 10% was considered as unrelated.*

4. STUDY II: EVALUATING PROMPTER WITH DEVELOPERS

The *goal* of this study is to evaluate to what extent the use of PROMPTER can be useful to developers during a development or maintenance task. The *quality focus* is the completeness (and correctness) of the task a developer can perform in a limited time frame, *e.g.,* because of a hard deadline. The *context* consists of *objects*, *i.e.,* participants have to perform two tasks with/without the availability of PROMPTER. We had 12 *participants* (3 BSc and 3 MSc CS students, and 6 industrial developers). Before the study, we screened the participants by using a pre-study questionnaire, asking them about their experience in programming and Java (The study tasks were in Java). The experience was measured in terms of (i) the number of years of Java programming, and (ii) a self-assessment based on a five-points Likert scale [27] going from 1 (very low experience) to 5 (very high experience). Also, we asked participants which sources of documentation they generally exploit when programming.

Participants. All participants have at least 3 years of experience in programming, with a maximum of 12 reached by an industrial developer and a median of 6.5. They have a median of 4 years of Java programming experience. Participants felt to have a good experience in programming and Java programming with a median of four (high experience) in both cases. Only two BSc students assessed their experience at 3 (medium), while all the others declared a high experience (4). The sources of information mostly exploited by participants when programming are: Stack Overflow (10 participants), Forums (8), Javadoc (8), and Books (6).

Tasks. The tasks participants have to perform are one mainte-

[3] `http://tinyurl.com/q4faaz5`
[4] `http://tinyurl.com/pxzpw6e`
[5] `http://tinyurl.com/orlyln3`

nance task and one greenfield development task (*i.e.,*, from scratch). The choice of tasks was performed taking into account that, being the study executed within a lab, the tasks could not be too long nor complicated. On the other side, the tasks could not be too simple, to avoid a "ceiling" effect, *i.e.,* that all participants correctly completed the tasks without problems, regardless of the use of PROMPTER.

Maintenance Task (MT). The maintenance task required the implementation of new features in a Java 2D arcade game, where the player controls a spaceship to destroy an attacking alien enemy fleet. We asked participants to perform the following changes: (i) add an initial screen from where players can start the game or look at top scores; (ii) allow players to enter their nickname before starting a game; (iii) when a game is over, save the score along with the player's nickname in a XML file; and (iv) allow a player to see the top 10 scores retrieved from the XML file.

Development Task (DT). For this task the participants had to create from scratch a Java program that, given the URL of a Web page and an e-mail address, converts the HTML page into a PDF and then send it via email. The task consisted in three sub-tasks: (i) allow the user to input the URL, e-mail address and subject of the email from text fields; (ii) retrieve the HTML page and convert it into a PDF; and (iii) send the PDF via e-mail.

We did not provide to participants any indication about the strategy to follow in the implementation of the two tasks, *e.g.,* we did not recommend the usage of any specific library.

4.1 Research Questions and Variables

The study addresses the following research question (RQ_2): *Does* PROMPTER *help developers to complete their task correctly?*

We aim at investigating if the use of PROMPTER helps developers when performing coding activities. We are interested in knowing to what extent—within the available time frame, and when working with or without PROMPTER —participants are able to correctly complete the task (or part of it).

The dependent variable aimed at addressing RQ_2 is the task completeness. Since it is difficult to automatically evaluate task completeness, we asked two independent industrial developers as "evaluators" to measure it by performing code reviews on each task implemented by participants. The evaluators did not know the goal of the study, nor which tasks were performed with (without) PROMPTER support. To help them in the assessment, we provided them with a checklist, aimed at assigning a fixed completeness score to each of the sub-tasks correctly implemented by participants when working on MT and DT. For example, for the maintenance task having correctly implemented each of the four sub-tasks would mean a completeness increment of 15%, 25%, 35%, and 25% respectively. The scores were proportional to the different difficulty and complexity of the four sub tasks. The detailed checklists are available in our replication package. The evaluators compared their independent, and conducted a discussion in case of diverging scores. This happened only on four out of the 24 evaluated tasks (*i.e.,* two tasks for each of the twelve participants) and the divergence was quickly solved by evaluators performing an additional code inspection.

The main factor and independent variable of this study is the presence or absence of PROMPTER. Specifically, such a factor has two levels, *i.e.,* the availability of PROMPTER (P) or not (NP). Other factors that could influence the results are (i) the (possible) different difficulty of the two tasks MT and DT, (ii) the participants' (self-assessed) *Ability* and (iii) *Experience* in Java development, and (iv) the years of *Industrial Experience* (if any) they may have.

4.2 Study Design and Procedure

The study design is a classical paired design for experiments with one factor and two treatments: (i) each participant worked with both P and NP; (ii) to avoid learning, each participant had to perform different tasks (MT and DT) across the two sessions; (iii) different participants worked with P and NP in different ordering, as well as on the two different tasks MT and DT. Overall, this means partitioning participants into four groups, receiving different treatments in the two laboratory sessions. When assigning participants to the four groups, we made sure that their level of experience was (roughly) uniformly distributed across groups.

Before the study, we conducted a pre-laboratory briefing, in which we trained participants on the use of PROMPTER, and illustrated the laboratory procedure. We made sure not to reveal the study research questions. In addition, the training was performed on tasks not related to MT and DT to not bias the experiment.

After that, the participants had to perform the study in two sessions of 90 minutes each. In other words, participants had a maximum of 90 minutes to complete each of the required tasks. Each participant received the instructions for the task she had to perform in the first session. After 90 minutes, each participant provided the code she implemented for the required task. Then, a 60 minutes break minutes was given before starting the second session to avoid fatigue effects. During the break participants did not have the chance to exchange information among them. After the break, each participant received the instructions for the task she had to perform in the second session. After 90 minutes, each participant provided the code she implemented for the required task. To simulate a real development context, participants were allowed to use whatever they want to complete the tasks including any material available on the Internet. After the study, we collected qualitative information by (i) using a post-study questionnaire, and afterwards, by (ii) conducting focus-group interviews.

The post-study questionnaire was composed of: (i) three questions asking if participants used Internet, the suggestions by PROMPTER, and their own knowledge during implementation. To answer these three questions participants used a four points scale, choosing between *absolutely yes*, *more yes than no*, *more no than yes*, and *absolutely no*; and (ii) a question asking participants to evaluate the relevance of the suggestions generated by PROMPTER. In this case, we adopted a five-points Likert scale [27] going from 1 (totally irrelevant) to 5 (very relevant).

During the focus-group interview, two of the authors and all participants discussed together about PROMPTER, trying to point out its weaknesses and strengths. This interview lasted 45 minutes.

4.3 Quantitative Analysis of the Results

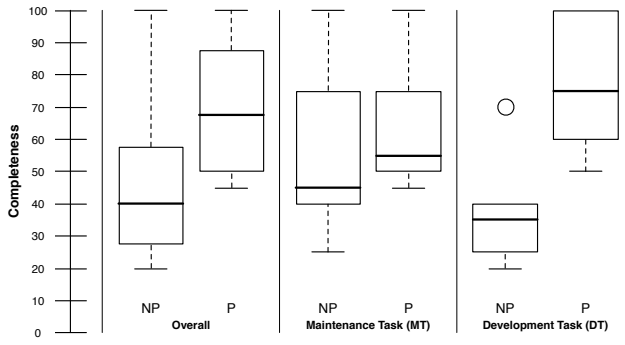

Figure 5: Boxplots of Completeness achieved by Participants with (P) and without (NP) PROMPTER**.**

Figure 5 shows boxplots of completeness achieved by participants when using (P) and when not using (NP) PROMPTER. As it can be noticed, participants using PROMPTER were able to achieve a level of completeness greater than those not using it. The P median is 68% (mean 70%) against the 40% median (mean 46%) of NP. In other words, PROMPTER allowed participants to achieve a median additional correctness of 28% (mean of 24%). Such a difference is also statistically significant. The Wilcoxon paired test indicates a p-value less than 0.01 for a significance level $\alpha = 5\%$. To assess the magnitude of such a difference, we used Cliff's delta (d) effect size [11], which returned a value of $d = 0.65$, *i.e.,* a *large* effect.

Figure 5 also shows boxplots of completeness when focusing on results achieved for *MT* and *DT*, respectively, to better understand where PROMPTER results particularly helpful. PROMPTER helped participants in both MT and DT, increasing the median completeness achieved for MT of 10%, and for DT of 40%. If we compare the completeness of P and NP for each one of the two tasks separately, Mann-Whitney unpaired, two-tailed test indicates that for MT the difference is not significant (p-value=0.23) and the effect size is 0.38 (*medium*), while it is statistically significant for DT (p-value=0.03), with an effect size of 0.88 (*large*). Since here multiple tests have been performed, p-values have been adjusted using Holm's correction [13]. PROMPTER produced much more benefits for DT, where participants implemented from scratch and where they had to use several libraries, *e.g.,* to parse the HTML page, to convert it in PDF, to send an e-mail. In such a circumstance, PROMPTER provided an effective support by pointing to Stack Overflow discussions concerning the correct usage of such libraries.

To check the influence of the various co-factors and their interaction with the main factor treatment, we used permutation test [5], a non-parametric alternative to Analysis of Variance (ANOVA).

Table 2: Effect of Co-Factors and their Interaction with the Main Factor: Results of Permutation Test.

Co-Factor	Effect	Interaction
Java Ability	**0.02**	1.00
Java Experience	**0.03**	0.68
Industry Experience	0.56	0.69
Task	0.81	0.10

The test p-values, summarized in Table 2, indicate that *Java Ability* and *Experience* have a significant effect on the participants' performance, although they do not interact with the main factor. In other words, people with higher ability and experience perform better, although independently on the availability of PROMPTER. Also, there is no effect nor interaction of the *Industry Experience*. Lastly, *Task* has no direct effect on the observed results, although it marginally interacts with the main factor. As one could have expected from Figure 5, PROMPTER resulted more helpful for DT than for MT.

In summary, the quantitative assessment provided a first indication of the usefulness of PROMPTER for developers performing maintenance activities (MT) or greenfield development (DT).

4.4 Qualitative Analysis of the Results

Results from the post-questionnaire provided us with interesting observations. Participants generally used Internet during the implementation of the required tasks. When being asked, 6 of them answered *absolutely yes*, 5 *more yes than no*, and 1 *more no than yes*. This is consistent with the answers they provided to the pre-study questionnaire. Second, most participants felt to have used their knowledge in the tasks implementation, with 4 of them answering *absolutely yes*, 6 *more yes than no*, and 2 *more no than yes*.

As for the question related to the use of PROMPTER's recommendations, most of participants answered positively. Three of them answered *absolutely yes*, eight *more yes than no*, and two *more no than yes*. The latter participants explained that they received very few PROMPTER recommendations, due to the fact that they spent much time on the Internet, trying to figure out how to implement the required tasks. This resulted in wasted time during which the code they were working on was untouched. Thus, PROMPTER was simply waiting in vain for their moves to produce suggestions. However, these two participants agreed that the few received recommendations were actually relevant to what they were implementing in the IDE. Among the twelve participants, two of them classified the suggestions as *very relevant* (5), and the remaining ten as *relevant* (4).

Apart from the questionnaire, we gained very useful insights from the focused group interview. On the one side, participants agreed that PROMPTER is very useful when working on tasks in which the developer has poor experience, since the information bring in the IDE by PROMPTER helps the developer in enriching her knowledge about the task to be performed. For instance, one of the participants was experiencing problems with the repaint function provided in the `JFrame` by the `updateUI` method. PROMPTER pushed in his IDE a Stack Overflow discussion[6] exactly related to what he was trying to implement, solving his problem. Another participant, when starting to work on DT, observed the push notification of PROMPTER about a Stack Overflow discussion[7] providing guidelines on how to choose the HTML parser library to use. After reading the discussion, his choice was targeted on `jsoup`.

Summarizing, on the one side participants identified the following PROMPTER strengths: (i) the accuracy of the suggestions and the relevance of the suggested Stack Overflow discussions; (ii) the user interface: clean, clear, and not invasive; (iii) the ease of use, with almost no training required; (iv) the possibility to tune the sensitivity of PROMPTER, increasing or reducing the rate of suggestions.

On the other side, they would like to see the following improvements in future PROMPTER releases: (i) the possibility to exploit information coming not only from Stack Overflow, but also from forums and programming tutorials available online; (ii) a way to force PROMPTER in looking for specific types of discussions on Stack Overflow. For example, participants would like the possibility to specify some key terms that should always be considered by PROMPTER when searching for discussions on Stack Overflow; and (iv) similarly, the possibility to have a search field: Most participants agreed on the fact that PROMPTER loses its usefulness if the developer has no idea on how to start coding. In such a situation, the developer is forced to leave the IDE and surf the Web. Participants suggested the addition of a search field in the PROMPTER user interface that allows one to explicitly formulate and execute a query without leaving the IDE.

Summary of Study II: PROMPTER *allowed participants to achieve a significantly better completeness of the assigned tasks. The collected feedbacks indicated that participants perceived the tool as usable, the suggestions accurate and not invasive. Eleven out of the twelve participants involved in our study claimed that they would like to use* PROMPTER *in their daily development activities. They also suggested to add to* PROMPTER *a feature to allow developers to explicitly formulate queries.*

5. THREATS TO VALIDITY

Threats to construct validity. In *Study I* such threats are due to (i) the fact that we mimic the code being written by a user by providing

[6] `http://stackoverflow.com/questions/11640494/`
[7] `http://stackoverflow.com/questions/3152138/`

with PROMPTER a partially-complete class, and (ii) by letting the users provide evaluations using a Likert scale. Concerning the former, we made sure such classes were not too detailed nor too empty, to represent realistic situations where PROMPTER could be used. Concerning the latter, this is a standardized evaluation scale used to collect participants' feedbacks. *Study II* overcomes the limitations of *Study I* mentioned above. In *Study II* threats to construct validity are due to how we measured the task completeness. Certainly, we could have used a test suite to measure the completeness in a objective manner. Conversely, code inspection allows to evaluate partial implementations. In addition, the use of a checklist and multiple independent evaluators limited the bias and subjectiveness.

Threats to internal validity. For *Study I* one factor to be considered is the knowledge of the participants—not known a-priori—of the APIs being used in the particular task. Having multiple participants with different degrees of experience mitigates this threat. Also, students taking part in our evaluation were not evaluated based on the task outcome, and we asked participants not to use other sources of information during the task, *e.g.,* to use them as a comparative source to the provided discussion. In *Study II*, to limit the effect of participants' ability and experience, we pre-assessed them to assign them to the four groups. We also analyzed to what extent the usefulness of PROMPTER depends on the particular task.

Threats to conclusion validity. For *Study I* we report descriptive statistics and violin plots of the collected results, along with participants' feedbacks. For *Study II*, we used distribution-free tests and effect size measures, suitable for limited data sets as in our study. The goal was to gain qualitative insights about the usefulness of PROMPTER, rather than to observe statistically significant results.

Threats to external validity. The study involved both professionals and students, with different degree of experiences. Therefore, we claim the study provides a good coverage of the potential categories of PROMPTER users, although further studies with more participants are desirable. In terms of objects, we selected 37 tasks being different in nature and required technical knowledge. We cannot exclude that our results depend on the particular choice of the tasks. For *Study II*, although we selected both students and industrial developers, it is worthwhile to replicate the study with a larger number of participants. Furthermore, PROMPTER was only evaluated with two tasks that, although different, are not representative enough for tasks that developers would perform. We believe that *Study I* achieves a better *external validity* whereas *Study II* a better *construct validity*.

6. RELATED WORK

PROMPTER is a recommender system that mixes different software engineering fields, namely code search and recommender systems.

Semantic code search and code search engines. The main usage of such search engines is to retrieve code samples and reusable open source code from the Web. Different works [31, 38, 39] tackled this problem and provided the developers with the capability of searching, ranking and adapting open source code. The mining of open source repositories has also been used to identify API and framework usage and to find highly relevant applications to be reused [25, 26, 40]. Other studies analyzed the usage and the habits of the developers in performing researches with code search engines (*e.g.,* Koders) [4, 22, 41], and how general purpose search engines (*e.g.,* Google) outperform code search engines when retrieving code samples from the Web [35]. In our work we follow an approach based on general-purpose search engines. Differently from the work done so far on code search, we do not target open source repositories to provide code samples and reusable code, or to understand the usage of APIs; instead, we target the crowd knowledge provided by the discussions in Stack Overflow as alternative source. This is because we want to provide developers with code examples with explanations, rather than just with reusable code components/snippets. The way PROMPTER interacts with search engines to retrieve discussions concerns code context analysis and matching. A milestone in identifying relevant code elements in the IDE is MYLYN [17]. The automation and generation of queries from code is another aspect that relates with PROMPTER. Holmes *et al.* [16, 15, 30] presented STRATHCONA, a tool to recommend relevant code fragments that automatically extracts queries from structural context of the code. In our work, we focus on the current element being modified by the developer. We take advantage of our own definition of context, and we apply an entropy-based approach to generate the query.

Recommender systems. Different typologies of recommender systems to recovery traceability links, suggest relevant project artifacts, and suggest relevant code elements in a project has been presented. Well-known examples are HIPIKAT [9], DEEPINTELLISENCE [14], and EROSE [43]. Other work focused on suggesting relevant documents, discussions and code samples from the web to fill the gap between the IDE and the Web browser. Examples are CODETRAIL [10], MICA [36], FISHTAIL [34], DORA [19], and SURFCLIPSE [29]. Among the various sources available on the Web, Q&A Websites and in particular Stack Overflow, have been the target of many recommender systems. Other tools used Stack Overflow as main source for recommendation systems to suggest, within the IDE, code samples and discussions to the developer [7, 32, 37, 29]. In our previous work we presented SEAHAWK [28], a prototype tool to link Stack Overflow discussions to the source code in the IDE using *tf-idf*. PROMPTER has several differences from SEAHAWK, that (i) does not rely on a ranking model mixing various factors, it just uses textual similarity; and (ii) it does not feature confidence-based push notifications, but rather it requires developers to explicitly trigger the queries. Finally, SEAHAWK has not been validated with developers.

PROMPTER is different from any recommender system proposed so far. PROMPTER is able to automatically and silently retrieve and rank Stack Overflow discussions relevant for the current code context. Then, it uses a configurable "self-confidence" mechanism to push suggestions, yet providing the developer with the possibility of consulting further relevant discussions whenever needed.

7. CONCLUSION AND FUTURE WORK

We have presented a novel approach to turn an Integrated Development Environment (IDE) into a developer's programming prompter. The approach is based on (1) automatically capturing the code context in the IDE, (2) retrieving documents from Stack Overflow, (3) ranking the discussions according to a novel ranking model we developed for this work, and (4) suggesting them to the developer when (and only if) it has enough self-confidence. We implemented our approach in a novel tool named PROMPTER, which embodies the ideal behavior a recommender should have: a silent observer of the developer, that only intervenes when it deems itself to have a relevant enough suggestion, and that does not force the developer to invoke it but is always available in case the developer needs it. Trough a quantitative study we showed how the PROMPTER ranking model resulted to be effective in identifying the right discussions given a code snippet to analyze. We evaluated PROMPTER during maintenance and development tasks. We showed how, from a quantitative point of view, PROMPTER revealed to significantly help developers in completing the experiment tasks and how, from a qualitative point of view, the developer highly appreciated its features and usability.

Acknowledgments Ponzanelli and Lanza thank the Swiss National Science foundation for the financial support through SNF Project "ESSENTIALS", No. 153129.

8. REFERENCES

[1] J. Anvik, L. Hiew, and G. Murphy. Who should fix this bug? In *Proceedings of ICSE 2006*, pages 361–370. ACM, 2006.

[2] A. Bacchelli, T. dal Sasso, M. D'Ambros, and M. Lanza. Content classification of development emails. In *Proceedings of ICSE 2012*, pages 375–385, 2012.

[3] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999.

[4] S. Bajracharya and C. Lopes. Mining search topics from a code search engine usage log. In *Proceedings of MSR 2009*, pages 111–120, 2009.

[5] R. D. Baker. Modern permutation test software. In *Randomization Tests*. Marcel Decker, 1995.

[6] L. Constantine. *Constantine on Peopleware*. Yourdon, 1995.

[7] J. Cordeiro, B. Antunes, and P. Gomes. Context-based recommendation to support problem solving in software development. In *Proceedings of RSSE 2012*, pages 85–89. IEEE Press, 2012.

[8] T. Cover and J. Thomas. *Elements of Information Theory*. Wiley-Interscience, 1991.

[9] D. Cubranic and G. Murphy. Hipikat: recommending pertinent software development artifacts. In *Proceedings of ICSE 2003*, pages 408–418. IEEE Press, 2003.

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

[11] R. J. Grissom and J. J. Kim. *Effect sizes for research: A broad practical approach*. Lawrence Associates, 2005.

[12] J. L. Hintze and R. D. Nelson. Violin plots: A box plot-density trace synergism. *The American Statistician*, 52(2):181–184, 1998.

[13] S. Holm. A simple sequentially rejective Bonferroni test procedure. *Scandinavian Journal of Statistics*, 6:65–70, 1979.

[14] R. Holmes and A. Begel. Deep intellisense: a tool for rehydrating evaporated information. In *Proceedings of MSR 2008*, pages 23–26. ACM, 2008.

[15] R. Holmes, R. Walker, and G. Murphy. Approximate structural context matching: An approach to recommend relevant examples. *IEEE TSE*, 32(12):952–970, 2006.

[16] R. Holmes, R. J. Walker, and G. C. Murphy. Strathcona example recommendation tool. In *Proceedings of ESEC/FSE 2005*, pages 237–240, 2005.

[17] M. Kersten and G. Murphy. Using task context to improve programmer productivity. In *Proceedings of FSE-14*, pages 1–11. ACM Press, 2006.

[18] A. J. Ko, R. DeLine, and G. Venolia. Information needs in collocated software development teams. In *Proceedings of ICSE 2007*, pages 344–353. IEEE CS Press, 2007.

[19] O. Kononenko, D. Dietrich, R. Sharma, and R. Holmes. Automatically locating relevant programming help online. In *Proceedings of VL/HCC 2012*, pages 127–134, 2012.

[20] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: a study of developer work habits. In *Proceedings of ICSE 2006*, pages 492–501. ACM, 2006.

[21] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Cybernetics and Control Theory*, (10):707–710, 1966.

[22] E. Linstead, P. Rigor, S. Bajracharya, C. Lopes, and P. Baldi. Mining internet-scale software repositories. In *In Proceedings of NIPS 2007*. MIT Press, 2007.

[23] L. Mamykina, B. Manoim, M. Mittal, G. Hripcsak, and B. Hartmann. Design lessons from the fastest q&a site in the west. In *Proceedings of CHI 2011*, pages 2857–2866. ACM.

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

[25] C. McMillan, M. Grechanik, D. Poshyvanyk, C. Fu, and Q. Xie. A source code search engine for finding highly relevant applications. *IEEE TSE*, 38(5):1069–1087, 2012.

[26] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu. Portfolio: finding relevant functions and their usage. In *Proceedings of ICSE 2011*, pages 111–120. ACM, 2011.

[27] A. N. Oppenheim. *Questionnaire Design, Interviewing and Attitude Measurement*. Pinter, London, 1992.

[28] L. Ponzanelli, A. Bacchelli, and M. Lanza. Leveraging crowd knowledge for software comprehension and development. In *Proceedings of CSMR 2013*, pages 59–66, 2013.

[29] M. Rahman, S. Yeasmin, and C. Roy. Towards a context-aware ide-based meta search engine for recommendation about programming errors and exceptions. In *Proceedings of CSMR/WCRE 2014*, page To appear, 2014.

[30] R. H. Reid and G. C. Murphy. Using structural context to recommend source code examples. In *Proceedings of ICSE 2005*, pages 117–125. ACM, 2005.

[31] S. Reiss. Semantics-based code search. In *Proceedings of ICSE 2009*, pages 243–253. IEEE, 2009.

[32] P. Rigby and M. Robillard. Discovering essential code elements in informal documentation. In *Proceedings of ICSE 2013*, pages 832–841, 2013.

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

[34] N. Sawadsky and G. Murphy. Fishtail: from task context to source code examples. In *Proceedings of TOPI 2011*, pages 48–51. ACM, 2011.

[35] S. Sim, M. Umarji, S. Ratanotayanon, and C. Lopes. How well do search engines support code retrieval on the web? *ACM TOSEM*, pages 1–25, 2011.

[36] J. Stylos and B. A. Myers. Mica: A web-search tool for finding api components and examples. In *Proceedings of VL/HCC 2006*, pages 195–202, 2006.

[37] W. Takuya and H. Masuhara. A spontaneous code recommendation tool based on associative search. In *Proceedings of SUITE 2011*, pages 17–20. ACM, 2011.

[38] S. Thummalapenta. Exploiting code search engines to improve programmer productivity. In *Proceedings of OOPSLA 2007*, pages 921–922. ACM, 2007.

[39] S. Thummalapenta and T. Xie. Parseweb: a programmer assistant for reusing open source code on the web. In *Proceedings of ASE 2007*, pages 204–213. ACM, 2007.

[40] S. Thummalapenta and T. Xie. Spotweb: Detecting framework hotspots and coldspots via mining open source code on the web. In *Proceedings of ASE 2008*, pages 327–336. IEEE, 2008.

[41] M. Umarji, S. Sim, and C. Lopes. Archetypal internet-scale source code searching. In *Proceedings of OSS 2008*, pages 257–263, 2008.

[42] L. Williams. Integrating pair programming into a software development process. In *Proceedings of CSEET 2001*, pages 27–36. IEEE, 2001.

[43] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *Proceedings of ICSE 2004*, pages 563–572. IEEE, 2004.