An Approach to Context-Based Recommendation in Software Development

Bruno Antunes
CISUC, University of Coimbra
Coimbra, Portugal
bema@dei.uc.pt

Joel Cordeiro
CISUC, University of Coimbra
Coimbra, Portugal
jfac@student.dei.uc.pt

Paulo Gomes
CISUC, University of Coimbra
Coimbra, Portugal
pgomes@dei.uc.pt

ABSTRACT

A software developer programming in an object-oriented programming language deals with a source code structure that may contain hundreds of source code elements. These elements are commonly related to each other and working on a specific element may require the developer to access other related elements. We propose a recommendation approach that uses the context of the developer to retrieve and rank recommendations of relevant source code elements in the IDE. These recommendations provide a shortcut to reach the desired elements and increase the awareness of the developer in relation to elements that may be of interest in that moment. We have tested our approach with a group of developers and the results show that context has a promising role in predicting and ranking the source code elements needed by a developer at each moment.

Categories and Subject Descriptors

D.2 [**Software Engineering**]: Programming Environments; H.3 [**Information Storage and Retrieval**]: Information Search and Retrieval; I.2 [**Artificial Intelligence**]: Knowledge Representation Formalisms and Methods

Keywords

Recommendation Systems, Context Modeling, Ontologies, Software Development, IDE.

1. INTRODUCTION

The recommendation systems are currently used in a wide variety of domains to help users find relevant information, deal with information overload and provide personalized recommendations of very different kinds of items. The recommendation process is commonly dependent on estimating the utility of a specific item for a particular user [1]. Most of the approaches used in current recommendation systems are mainly focused in estimating how relevant is an item to an user, ignoring any contextual information that could be used to improve the recommendation process. However, context-based recommendation systems are emerging, taking context into account when providing recommendations to the user [2].

A recommendation system for software engineering has been defined by Robbilard et al. [10] as *"a software application that provides information items estimated to be valuable for a software engineering task in a given context"*. The increasing dimension and complexity of software development projects are fostering the development of such systems, which have been applied to very different tasks such as software reuse, expertise location, code comprehension, guided software changes, debugging, etc.

A software developer programming in an object-oriented programming language, such as Java, deals with a source code structure that may contain hundreds of source code elements. These elements are commonly related to each other and working on a specific element may require the developer to access other related elements. Despite some features that allow the developer to jump between related elements in very specific situations, the primary way to access these elements is by browsing the source code structure, either using the package structure or the outlines provided for individual source code files. As the number of source code elements in the workspace of a developer increase, the need to switch between different elements becomes more frequent and more time is spent in finding the needed elements. We propose a recommendation approach to help developers programming in Java, by recommending source code elements that are considered relevant in the context of the developer at a specific moment. These recommendations provide a shortcut to reach these elements, and a way to increase the awareness of the developer in relation to elements that may be of interest in that moment. The recommendations are provided in a pro-active way and automatically updated as the context of the developer changes.

The recommendation approach uses the context of the developer to retrieve and rank recommendations of source code elements, such as classes, interfaces and methods, in an IDE (Integrated Development Environment). These elements and their relations are represented in a knowledge base using an ontology [15]. The contextual information is implicitly gathered from the interactions of the developer in the IDE, and is used to retrieve and rank relevant source code elements that are then recommended to the developer. We have implemented a prototype, named SDiC[1] (Software Development in Context), that integrates our recommenda-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
RecSys'12, September 9–13, 2012, Dublin, Ireland
Copyright 2012 ACM 978-1-4503-1270-7/12/09 ...$15.00.

[1]http://sdic.dei.uc.pt

tion approach in an Eclipse[2] plugin. This prototype was submitted to a field experiment with a group of developers. The results show that the use of the context has a promising role in predicting and ranking the source code elements needed by a developer on a specific moment.

The remainder of the paper is organized as follows. In section 2 we make an overview of related work. Then, a description of the knowledge based used is given in section 3. In section 4 we introduce the context model and section 5 explains how this model is used to support the recommendation process. The prototype implemented is presented in section 6, followed by evaluation and result discussion in section 7. Finally, section 8 concludes the work with some final remarks and future work.

2. RELATED WORK

The context of a developer has been modeled in different ways to improve awareness and help locating relevant source code elements through recommendation. Kersten and Murphy [9] propose a model for representing the context associated to a task. The task context is used to help focus the information displayed in the IDE and to automate the retrieval of relevant information for completing the task. Our approach also uses the interactions of the developer to derive the degree of interest of different source code elements to the developer, which is used together with an ontological model of the source code structure to retrieve and rank other relevant source code elements. The context model we use is not attached to tasks and automatically adapts to the changes in the focus of attention of the developer. Warr and Robillard [12] developed Suade, a plugin for the Eclipse IDE that help developers exploring the source code, by suggesting potentially relevant elements for the current context. The context is explicitly provided by the developer and the suggestions are ranked according to their structural dependencies using some heuristics. We do not require an explicit definition of the current context to provide such recommendations, as it is automatically gathered by the system. Heinemann and Hummel [7] propose an approach to recommend methods that may be relevant for the current work of a developer. The recommendations are based on the terms extracted from identifiers preceding method calls. We also explore the lexical dimension of source code elements, which is integrated in our context model and then used to rank the recommendations provided to the developer.

The contextual information was also used to improve the recommendation of reusable software components or useful source code examples. Ye and Fischer [13] propose CodeBroker, a tool that pro-actively suggests reusable components to Java developers using Emacs. The system monitors the JavaDoc comments and signature definitions created by the developer, from which it extracts queries to retrieve matching components. Strathcona, an Eclipse plugin proposed by Holmes and Murphy [8], is used to help developers locating source code examples that are relevant for their current task. When the developer requests for examples related with a class, method or field declaration, the system generates a structural description of these elements to find source code examples that match that description. PARSEWeb was proposed by Thummalapenta and Xie [11] to suggest examples of method call sequences, to help developers going from a specific object type to a desired object type. These approaches focus on recommending external source code elements that can be reused or used as examples to help developers in their current task. We propose an approach to help developers locating relevant source code elements that exist in their workspace.

Some works retrieve recommendations of relevant artifacts based on the contextual information provided by the history associated to a software development project. Hipikat was developed by Cubranic et al. [5] to recommend relevant artifacts from a project's history. The project memory is built from the various kinds artifacts created during a software development project. These artifacts are recommended to the developer when they are considered relevant for a task being performed. eRose [14], presented by Zimmermann et al., provide recommendations of source code elements that were changed together. The changes applied to the source code are mined from a CVS archive and used to predict further changes, reveal hidden couplings and avoid errors produced by incomplete changes. Our approach is focused in the context of the developer at each moment and do not take into account historical information.

3. KNOWLEDGE BASE

The recommendation system is based on a knowledge base that uses an ontology to represent of the source code structure that is stored in the workspace of a developer (see figure 1). This knowledge base is unique for each developer, being built from the source code files with which the developer is working, and maintained as these files are changed. An earlier version of this model has been used to support the search of source code elements in the workspace [3]. The previous model has been extended with additional relations and some statistical information. The source code structure is represented using two top-level elements, one of them representing structural elements and other representing lexical elements.

The structural elements include classes, interfaces and methods. These elements are connected by a set of relations that represent inheritance (*extensionOf* and *implementationOf*), composition (*attributeOf* and *methodOf*) and behavior (*parameterOf*, *returnOf*, *calledBy* and *usedBy*). This way, we create a representation of the source code structure used in the Java programming language, where the static relations between the different elements become explicit.

The lexical elements represent terms that are extracted from the names of the structural elements. The naming convention used in the Java programming language, known as CamelCase, provides a way to distinguish between the various terms used to create a composed name. These terms are extracted from the name of the structural elements represented in the knowledge base and become associated to those elements using an *indexedBy* relation. Each term can be associated to more than one structural element, because it may be used to compose the name of different elements. The number of times a term is used to index a structural element represents its frequency in the knowledge base and is also stored. When two terms are used together to compose the name of a structural element we create an *associatedWith* relation between them. This relation is used to represent the proximity between the terms, the same way co-occurrence is interpreted as an indicator of semantic proximity in linguistics [6]. The number of times the two terms co-occur

[2]http://eclipse.org

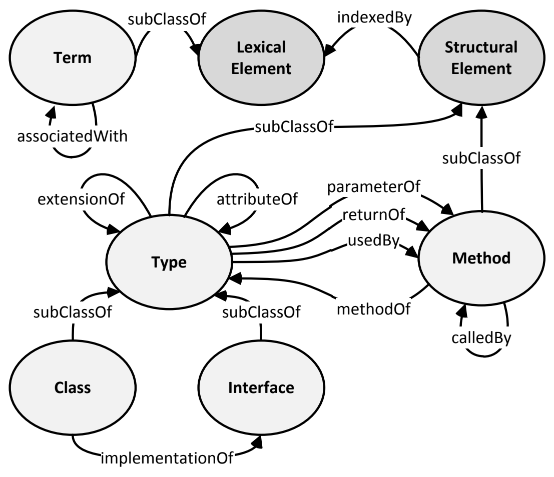


Figure 1: The knowledge base model.

in the names of different structural elements is stored and used as a weight associated to the relation. Terms that cooccur more often have a stronger relation than those that are rarely used together. The lexical elements, and their relations, can be used to find source code elements that are related by the terms that compose their names.

4. CONTEXT MODEL

The context model is based on the source code elements that are more relevant for developers during their work, providing an insight to what is their focus of attention at each moment. The model automatically adapts to the behavior of developers by detecting changes in their focus of attention. This model has been studied before and described in more detail in [4]. The contextual information is implicitly gathered from the interactions of the developer with the source code elements in the workspace. These elements and their relations are represented as the structural context, while the terms used to compose the names of these elements are represented as the lexical context. The relevance of each element is given by a Degree of Interest (DOI), a concept previously introduced by Kersten and Murphy [9].

When the developer interacts with the source code elements in the workspace, these elements are added to the structural context and their DOI is increased according to the type of interaction. When the developer closes or stops interacting with a source code element, its DOI is decreased until it is removed. Based on the source code elements that are more relevant for the developer, we also extract the relevance of the structural relations that are represented in the knowledge base. The relations that exist between the source code elements in the structural context are considered more relevant for the developer. The relevance of these relations is also given by a DOI, which is computed as an average of the DOI of the source code elements that are connected by that relation. The terms extracted from the names of the source code elements that exist in the structural context are added to the lexical context. The relevance of each term is given by its DOI, which is computed as an average of the DOI of the source code elements from which it was extracted.

For instance, imagine a situation where the developer interacts with a class named *ContextModel* and a method of that class named *getContext*. These two elements are added to the structural context and their DOI will reflect the interactions of the developer with that elements over time. Because the two elements are bound by a *methodOf* relation, this relation is also added to the structural context and its DOI is computed as an average of the DOI of the two elements. The terms that comprise the name of the elements, which are *context*, *model* and *get* are added to the lexical context with a DOI computed as an average of the DOI of the elements from where they were extracted.

5. RECOMMENDATION

We have implemented a process that uses the context model described before to retrieve and rank relevant source code elements that are recommended to the developer. Our aim is to help developers reaching the desired source code elements more easily and quickly, decreasing the effort needed to search for that elements in the entire source code structure. This process is based on the fact that most of the source code elements needed by the developer are likely to be structurally and lexically related with the elements that are being manipulated in that moment [4]. The context model plays a central role in this process, providing a mechanism to identify and evaluate the relevance of the source code elements that are being manipulated. We use the structural elements represented in this context model, along with their relations, to retrieve recommendations of elements that are potentially relevant for the developer. These recommendations are then ranked taking into account different components, that represent the retrieval process and the proximity to the context model. Finally, a feedback mechanism allows the system to learn which components are more relevant for the developer and adapt the ranking algorithm in order to favor that components.

5.1 Retrieval

The recommendations are retrieved using the source code elements in the context model with two different approaches, one based on the DOI of the elements and other based on the time elapsed since the elements were last accessed. The first emphasizes the overall importance of the source code artifacts in the context of the developer, while the later privileges the artifacts that were recently accessed by the developer.

The DOI based approach makes use of the relevance of the source code elements that have been manipulated by the developer to identify potentially relevant elements. The recommendations include the top N elements with higher DOI in the context model and all the elements that are structurally related with them. We call N the query size of the recommendation process, and the value of N with which we have achieved the best results is 3 (see section 7), which is the default value used by the system. But this value can also be defined by the developer in specific situations.

The time based approach uses the time, instead of the DOI, to measure the relevance of the source code elements that are being manipulated by the developer. The DOI of an element represented in the context model reflects the relevance of that element during a period of time. But, sometimes, the most relevant elements may not be those with an higher DOI during that period of time, but the ones that have been accessed more recently. The time based approach

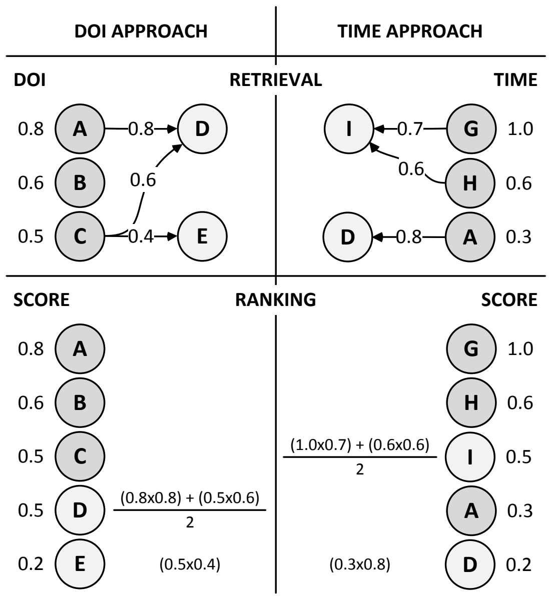


Figure 2: Representation of the retrieval and ranking using the DOI and time based approaches.

favors this aspect, retrieving source code elements that are related with the the elements that have been manipulated more recently. The recommendations retrieved using this approach include the top N elements of the context model that have been accessed more recently and all the elements that are structurally related with them.

A simplified representation of the retrieval process is shown in the upper side of figure 2. The DOI based approach starts by collecting the top 3 source code elements with higher DOI in the context model, shown as elements A, B and C. Then, all the elements that are related with these elements are also retrieved, which are represented by elements D and E. The time based approach first retrieves the 3 elements that were accessed more recently by the developer, represented as elements G, H and A, and then all the elements that are related with them, represented by elements I and D. The values associated with elements A, B and C represent their DOI in the context model. The values associated with elements G, H and A represent the time elapsed since their were accessed, normalized by the time of the most recently accessed element, which is element G. The values associated with the relations represent the DOI of these relations in the context model.

5.2 Ranking

The recommendations retrieved are then ranked taking into account the retrieval process and their proximity to the context model. The final score of each recommendation, which determines its ranking, is computed using four components: DOI, time, structural and lexical. The first two components represent the scores associated with the retrieval process, while the other two represent the scores of the recommendation in relation to the context model. The final score of a recommendation is given by a weighted sum of these scores.

The DOI score represents the score of the elements that were retrieved using the DOI based approach. There are two types of elements retrieved, those that are in the list of the elements with higher DOI in the context model, and the ones that are structurally related with them. The elements that are retrieved in the list have a score that corresponds to their DOI in the context model. The score of the elements retrieved through a structural relation with the elements in the list is computed using the DOI of the relation and the DOI of the element with which they are related. The DOI of the element in the list is used to normalize the DOI of the structural relation, so that the score of the retrieved element is proportional to the DOI of the element in the list. This way, the score of the retrieved elements take into account the relevance of both the relation and the element that contributed to their retrieval. When an element have a structural relation with more than one of the elements in the list, the score is given by the average of the scores of all the relations. The time score represents the score of the elements that were retrieved using the time based approach. The score is computed in a way that is similar to that used to compute the DOI score. The main difference is that the relevance of each element is computed using the time elapsed since it was last accessed, instead of using its DOI. This time span is normalized by the maximum time span among the elements in the top N list.

As represented in the lower side of figure 2, the elements with higher DOI retrieved in the top N list (A, B and C) have a DOI score correspondent to their DOI in the context model (respectively 0.8, 0.6 and 0.5). Element E was retrieved using a structural relation with element C, thus its DOI score is computed by normalizing the DOI of that relation (0.4) with the DOI of element C (0.5). Finally, element D was retrieved using two structural relations with elements A and C, and its score is computed as an average of the DOI of these relations normalized by the DOI of elements A and C. The same rules apply to the elements retrieved using the time based approach, using the normalized time instead of the DOI of the retrieved elements.

The structural score represents the proximity of the recommendation in relation to the structural context. We define this proximity as a distance between the source code element being recommended and the elements in the structural context. The distance between two source code elements is computed using the structural relations represented in the knowledge base and the relevance of these relations in the structural context. The structural relations allow us to find paths between the source code elements. Instead of using a fixed cost for each relation, the cost of a relation is inversely proportional to the DOI of that relation in the context model. The total cost of a path is given by the sum of the cost of the relations that create the path. This way we take into consideration the current relevance of the structural relations to the developer, assuring that the paths created with relevant relations will have a lower cost. The distance between two source code elements is given by the minimum path cost between those elements. The proximity of the source code element being recommended to the elements in the structural context is computed as an average of the minimum path costs between these elements. We only consider the top 15 elements with higher DOI in the structural context and paths with a maximum of 3 relations, to assure an acceptable performance. The average path cost is normalized, using equation (1), and the structural score is then given by inverting the normalized path cost. The normalization is used to obtain a score value in the interval $[0, 1]$.

The lexical score represents the proximity of the recommendation in relation to the lexical context. We define this proximity as a distance between the terms that comprise the name of the source code element being recommended and the terms in the lexical context. The distance between two terms is computed using the *associatedWith* relations and their weights, which are represented in the knowledge base. These relations allow us to find paths between terms. The cost of these paths is inversely proportional to the weight of the relations that create the path. The weight of each relation represents the frequency of co-occurrence of the two terms and is normalized by the maximum co-occurrence frequency in the knowledge base. By using the weight of the relations between terms to compute the cost of a path, we assure that the paths between terms that co-occur more frequently will have a lower cost. We have also identified a set of terms (such as *get*, *set*, *add*, *to*, *is*, etc) that appear very frequently in the name of the source code elements, especially methods. These very frequent terms co-occur with a variety of other terms and end up connecting almost every term in a distance of a few relations, thus distorting our metric. This problem could be partially solved with a list of very frequent terms that should be ignored, but this would be very limiting, because the top frequent terms vary from workspace to workspace and may include terms that are specific to each workspace. This way, we chose to ignore all the terms that would fall in the top 30% of all term occurrences. The paths that go through one of these terms are discarded. This percentage is based on our observation of a group of knowledge bases from different users, but needs to be further studied. The proximity of the source code element being recommended to the terms in the lexical context is computed as an average of the minimum path costs between terms. Again, we only consider the top 15 terms with higher DOI in the lexical context and paths with a maximum of 3 relations. The average path cost is normalized, using equation (1), and the lexical score is then given by inverting the normalized path cost.

$$1 - \left(\frac{1}{e^x}\right) \tag{1}$$

5.3 Learning

We have implemented a learning mechanism that uses the recommendations that have been selected by the developer to learn the weights that are associated to the components used in the ranking process. This is not used as a collaborative filtering mechanism, as the learning is limited to the weights used for an individual developer. We use the ranking obtained by the recommendation for each component to find the influence of that component in the final ranking. The final ranking of a recommendation represents its place in the list of all recommendations sorted by their final score, while the ranking of a component represents the place of the recommendation when the list is sorted by the score of that component. The objective is to increase the weights of the components that contribute to promote recommendations and decrease the weights of the components that contribute to demote them.

When the developer selects a recommendation which ranking was influenced by two or more components, the learning process is initiated. As shown in equation (2), we compute the influence of each component (i_x) using the difference between the final ranking of the recommendation (rf) and the individual ranking obtained for that component (rx). This way, the higher a recommendation is ranked in a specific component, the higher is the influence of that component in the final ranking. The influence obtained for each component is then normalized to the interval $[-1, 1]$, so that the components that had a higher influence get positive values, while those that had a lower influence get negative values. The normalized influence (ni_x) is given by equation (3), where i_{min} and i_{max} represent the minimum and maximum influence among all components.

$$i_x = rf - rx \tag{2}$$

$$ni_x = 2 \times \left(\frac{i_x - i_{min}}{i_{max} - i_{min}}\right) - 1 \tag{3}$$

Then, the positive and negative influences must be balanced, so that the weights are increased in the same proportion they are decreased, maintaining their sum as 1. The balanced influence (bi_x) is given by equations (4) and (5), where ni_t represents the sum of the influences in each group and m represents the number of components in each group.

$$bi_x^+ = \frac{ni_x^+}{ni_t^+}; \quad ni_t^+ = \sum_{k=0}^{m^+} ni_k^+ \tag{4}$$

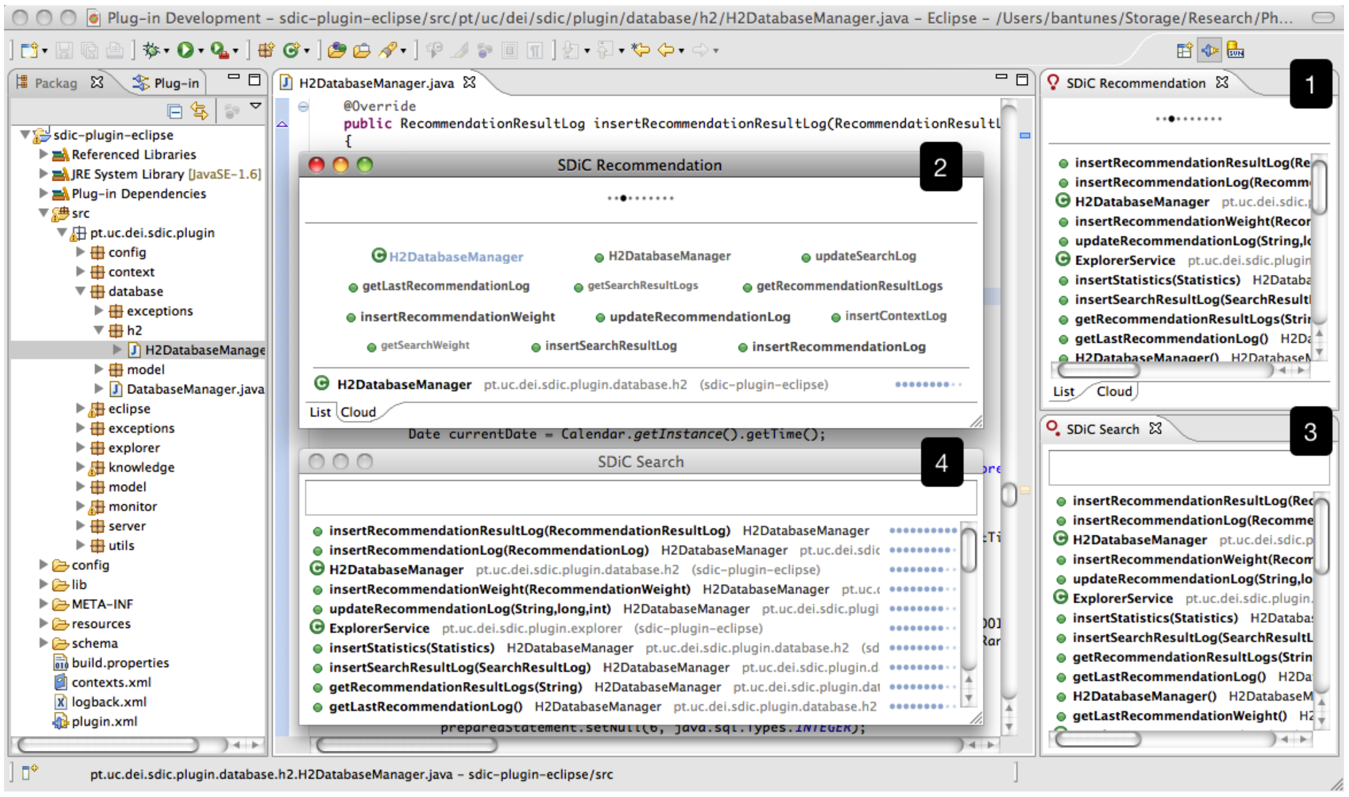
$$bi_x^- = \frac{|ni_x^-|}{ni_t^-}; \quad ni_t^- = \sum_{k=0}^{m^-} ni_k^- \tag{5}$$

Because the learning effort needed by the system must depend on how correct is the recommendation, a learning coefficient is applied to the balanced influence. The value of the learning coefficient depends on the final ranking of the recommendation that was selected by the developer. The better the ranking of the recommendation result, the lower the learning effort needed. This way, the learning coefficient (μ) is given by equation (6), where rf is the final ranking of the recommendation and 0.01 is the maximum value for the learning coefficient. Finally, the difference for each weight is obtained by applying the learning coefficient to the balanced influence. The new weight is obtained by adding this difference to the previous weight.

$$\mu = 0.01 \times \left(1 - \left(\frac{1}{e^{0.2 \times rf}}\right)\right) \tag{6}$$

6. PROTOTYPE

We have developed a prototype that integrates our recommendation approach in Eclipse using a plugin. The recommendations are available through a specific interface, that can be used as an Eclipse View (see 1 in figure 3) or a Window (see 2 in figure 3). The Eclipse View can be integrated in the main window of the IDE and stay visible all the time to keep the recommendations easily accessible. The Window is an alternative that spares space in the IDE, it can be triggered using a combination of keys and is fully functional using only the keyboard. This interface provides recommendations in the form of a list and in what we call a



```
@Override
public RecommendationResultLog insertRecommendationResultLog(RecommendationResultL
{
    Date currentDate = Calendar.getInstance().getTime();
    preparedStatement.setNull(6, java.sql.Types.INTEGER);
```

```
H2DatabaseManager  pt.uc.dei.sdic.plugin.database.h2  (sdic-plugin-eclipse)
```

```
insertRecommendationResultLog(RecommendationResultLog)  H2DatabaseManager
insertRecommendationLog(RecommendationLog)  H2DatabaseManager  pt.uc.dei.sdic
H2DatabaseManager  pt.uc.dei.sdic.plugin.database.h2  (sdic-plugin-eclipse)
insertRecommendationWeight(RecommendationWeight)  H2DatabaseManager  pt.uc.d
updateRecommendationLog(String,long,int)  H2DatabaseManager  pt.uc.dei.sdic.plugi
ExplorerService  pt.uc.dei.sdic.plugin.explorer  (sdic-plugin-eclipse)
insertStatistics(Statistics)  H2DatabaseManager  pt.uc.dei.sdic.plugin.database.h2  (sd
insertSearchResultLog(SearchResultLog)  H2DatabaseManager  pt.uc.dei.sdic.plugin.d
getRecommendationResultLogs(String)  H2DatabaseManager  pt.uc.dei.sdic.plugin.dat
getLastRecommendationLog()  H2DatabaseManager  pt.uc.dei.sdic.plugin.database.h2
```

```
insertRecommendationResultLog(Re
insertRecommendationLog(Recomm
H2DatabaseManager  pt.uc.dei.sdic.
insertRecommendationWeight(Recor
updateRecommendationLog(String,l
ExplorerService  pt.uc.dei.sdic.plugin
insertStatistics(Statistics)  H2Databa
insertSearchResultLog(SearchResult
getRecommendationResultLogs(Strir
getLastRecommendationLog()  H2Da
H2DatabaseManager()  H2DatabaseM
```

```
insertRecommendationResultLog(Rec
insertRecommendationLog(Recomme
H2DatabaseManager  pt.uc.dei.sdic.p
insertRecommendationWeight(Recom
updateRecommendationLog(String,lo
ExplorerService  pt.uc.dei.sdic.plugin.
insertStatistics(Statistics)  H2Databa
insertSearchResultLog(SearchResultL
getRecommendationResultLogs(Strin
getLastRecommendationLog()  H2Da
H2DatabaseManager()  H2DatabaseM
getLastRecommendationWeight()  H2
```

Figure 3: A screenshot of Eclipse running the prototype.

code cloud (see 2 in figure 3). The recommendations are automatically updated as the developer interacts with the IDE, reflecting the changes in the context model. The number of elements of the structural context that are used for the retrieval process, which we call of query size, can be defined by the developer, between 1 and 10, and is shown in the dotted scale in the top of the interface. A maximum of 30 recommendations are presented to the developer each time the recommendations are updated. The prototype also provides a search interface (see 3 and 4 in figure 3), which will not be discussed here, but is important to understand the relevance of our approach. The recommendations were integrated with this interface and are automatically shown to the developer before performing the search. We are assuming that if the recommendations include the source code element desired by the developer, this will avoid the need to perform the search. The query size used for the recommendations shown in the search interface is 3, which is the query size that obtained the best results in our evaluation (see section 7).

7. EVALUATION

We have created an experiment to validate our approach in the field, with developers using the prototype during their work. The experiment was conducted with a group of 5 developers from industry and 10 developers studying computer science, all of them using Eclipse to develop source code in the Java programming language. The application was installed in their work environment and presented as a tool to recommend source code elements inside the IDE. They have used the application for different periods of time, ranging from 2 weeks to 3 months. The objective of this experiment was to show that our recommendation approach would help developers find the necessary source code elements more quickly and efficiently. More specifically, we wanted to prove that recommendations could be used to avoid the need to perform a search or browse the source code structure to find the needed elements. We also wanted to prove that the context of the developer could be used to identify the most relevant source code elements for the developer and that it would have a positive impact on the ranking of these elements. The results analyzed here refer to the recommendation process only, although being occasionally compared with the search results, for the sake of comprehension.

7.1 Quantitative Study

We wanted to evaluate the capacity of the system in predicting the source code elements that the developers would need in the near future, so that these elements could be proactively recommended to them. Also, we wanted to discover what query size should be used to achieve the better results. This evaluation was performed in the background, by verifying if the source code elements being opened for the first time were already being recommended by the system. This way, we were able to evaluate our approach using the behavior of the developers during their work, without requiring them to use our recommendations. We have implemented a mechanism to store the top 30 recommendations generated by the system with a random query size (between 1 and 10).

Table 1: Percentage of opened source code elements found in recommendations, and average rankings, per query size.

QS	T	P	F	RD	RT	S	L
1	775/1897	40.9%	6.4	7.6	6.4	7.0	5.5
2	998/1867	53.5%	10.0	10.5	10.9	9.7	8.1
3	985/1807	54.5%	12.6	11.7	14.9	11.4	9.7
4	920/1854	49.6%	14.3	13.8	17.5	12.6	10.4
5	863/1838	47.0%	14.6	13.6	18.7	12.1	10.7
6	763/1842	41.4%	15.3	13.9	19.9	12.1	11.5
7	687/1808	38.0%	14.9	13.5	20.2	11.8	11.5
8	630/1770	35.6%	14.8	14.1	20.3	11.8	11.4
9	637/1845	34.5%	14.7	13.6	20.4	11.2	11.2
10	570/1799	31.7%	15.0	14.3	20.1	11.8	11.0
Total	7828/18327	42.7%	13.1	12.5	16.5	11.1	10.0

For each source code element opened for the first time, we have verified if that element was being recommended by the system in that moment. In table 1, we present the results obtained per each query size, where *QS* represent the query size, *T* the total number of elements found in recommendations, *P* the percentage of elements found, *F* the average final ranking, *RD* the average retrieval DOI ranking, *RT* the average retrieval time ranking, *S* the average structural ranking and *L* the average lexical ranking. In average, considering all query sizes, 42.7% of the source code elements opened for the first time were already being recommended by the system. The best results were achieved with a query size of 3, with which the system has been able to predict the developer needs in 54.5% of the times. As expected, the results also show that very lower query sizes tend to have worse values, as the number of source code elements used to retrieve the recommendations is not enough to reach the desired element. The higher query sizes also have worse results, which can be explained by the fact that when we increase the number of source code elements in the retrieval process, the recommendations became more dispersed and the probability of finding what the developer needs decreases. We believe that these results are very interesting and show that the context of developers has much to say about their immediate needs. Although the rankings can still be improved, the results also show a tendency for better rankings obtained with the context components, which reveals a positive impact of the context model in the final ranking.

We have also collected information about the recommendations selected by the developers. A total of 348 recommendations were selected. Almost 98% were selected from the recommendations integrated in the search interfaces, which means that the developer had the intention to search for a specific source code element, but the desired element was recommended even before performing the search. When we take into account all the source code elements selected, both from search and recommendation, the recommendations represent about 35% of all selected elements. This means that in 35% of the times in which the developer used our prototype to reach a desired source code element, the need of the developer was satisfied by a recommendation. These numbers are a good indication that our recommendation approach is reducing the effort of the developer on finding the needed source code elements, avoiding the need to explore the source code structure or perform a search to find these elements. The recommendation exclusive interfaces were almost ignored, especially the *code cloud*, which can be explained by the fact that an interface that provides proactive recommendations and also allows to perform a search delivers more value to the developer.

The average final ranking of the selected recommendations was 5.4. Although it may still be subject to improvements, we consider that this is a good precision for a recommendation system. The average rankings of the individual components were 9.3 for the DOI component, 9.2 for the time component, 6.6 for the structural component and 5.4 for the lexical component. There is a clear difference between the retrieval and the context components, with the context components obtaining better rankings, showing that the use of context is having a positive impact in the ranking of recommendations. Interestingly, the lexical component achieves better rankings than the structural one, which depicts that the lexical relations between source code elements are contributing to improve the precision of the recommendations. This way, we believe that context should be used not only to retrieve the recommendations, but also to distinguish which can be more relevant to the developer.

7.2 Qualitative Study

The developers were requested to fill an anonymous questionnaire to give their opinion about the recommendations provided by the system. We collected results from 10 out of the 15 developers that used the prototype, which are presented in table 2. The usability of the system was considered good and the relevance of recommendations was rated positively. The utility of the recommendations and the learning mechanism were also rated above the average. The raking of recommendations is something that can be improved and the impact in the productivity could not be verified.

We also asked developers about the things they liked the most and the least, and which suggestions they would give to improve the system. Most of the developers said the recommendations were an easy, fast and useful way to jump into the desired source code elements, avoiding the need to investigate the source code structure, look inside source code files or even perform a search. A few of them said that performing a search was sometimes faster than looking for the desired elements in the list recommendations. Thus, the interface is something that could be improved, in order to find more interesting ways of presenting the recommendations to developers. The capacity of the system in dealing with context transitions is something that could also be improved. Other improvements suggested include using the recommendations in code-completion and to highlight relevant methods directly in the source code.

8. CONCLUSION

We have presented an approach to recommendation in software development that uses a context model of the developer to retrieve and improve the ranking of source code elements that are recommended to the developer. A prototype was experimented by a group of developers. The results obtained are interesting, showing a promising capacity of our recommendation approach in predicting the developer needs, which can be used to reduce the effort of the developer on finding the needed source code elements. The use

Table 2: Questionnaire results per question.

QUESTION	SCALE	AVG	SD
How would you rate the utility of the recommendation functionality?	Very Low (1) - (5) Very High	3.8	0.93
How would you rate the usability of the recommendation functionality?	Very Poor (1) - (5) Very Good	4.3	1.09
How would you rate the impact of the recommendation functionality in your productivity?	Very Low (1) - (5) Very High	3.2	1.26
How would you rate the overall relevance of recommendations?	Very Irrelevant (1) - (5) Very Relevant	4.0	0.45
How often did relevant recommendations appear among all recommendations?	Very Rarely (1) - (5) Very Often	4.0	1.18
How often did relevant recommendations appear well ranked among all recommendations?	Very Rarely (1) - (5) Very Often	3.6	1.16
How would you rate the improvement in ranking of relevant recommendations over time?	Very Low (1) - (5) Very High	3.8	1.71

of the context model is crucial to identify relevant elements and assess their relevance, having a positive impact in the ranking of the recommendations provided to the developer. As future work we plan to improve the recommendation process, especially with regard to accuracy. We need to evaluate the impact of each individual component in the final ranking of recommendations and understand how the retrieval process can be improved to increase the coverage of the algorithm. Because the recommendation of relevant source code elements in the IDE is something the developers are not used to, the interface needs to be improved, seeking innovative ways for providing such recommendations. Also, it is important to evaluate how the time saved using the recommendations compare with other approaches, such as search. Although the results obtained are very promising, there is space for improvements and the approach needs to be validated with a wider range of developers.

9. ACKNOWLEDGMENTS

Bruno Antunes is supported by the FCT scholarship grant SFRH/BD/43336/2008, co-funded by ESF (European Social Fund).

10. REFERENCES

[1] G. Adomavicius and A. Tuzhilin. Toward the next generation of recommender systems: a survey of the state-of-the-art and possible extensions. *Knowledge and Data Engineering, IEEE Transactions on*, 17(6):734 – 749, june 2005.

[2] G. Adomavicius and A. Tuzhilin. Context-aware recommender systems. In F. Ricci, L. Rokach, B. Shapira, and P. B. Kantor, editors, *Recommender Systems Handbook*, pages 217–253. Springer US, 2011.

[3] B. Antunes, J. Cordeiro, and P. Gomes. Context-based search in software development. In *Proc. of the 7th International Conference on the Prestigious Applications of Intelligent Systems (Accepted for publication)*, Montpellier, France, August 2012. IOS Press.

[4] B. Antunes, J. Cordeiro, and P. Gomes. Context modeling and context transition detection in software development. In *Proc. of the 7th International Conference on Software Paradigm Trends (Accepted for publication)*, Rome, Italy, July 2012.

[5] D. Cubranic, G. C. Murphy, J. Singer, and K. S. Booth. Hipikat: A Project Memory for Software Development. *IEEE Transactions on Software Engineering*, 31(6):446–465, 2005.

[6] Z. Harris. Distributional structure. *Word*, 10(23):146–162, 1954.

[7] L. Heinemann and B. Hummel. Recommending api methods based on identifier contexts. In *Proceedings of the 3rd International Workshop on Search-Driven Development: Users, Infrastructure, Tools, and Evaluation*, SUITE '11, pages 1–4, New York, NY, USA, 2011. ACM.

[8] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 117–125, New York, NY, USA, 2005. ACM.

[9] M. Kersten and G. C. Murphy. Using task context to improve programmer productivity. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 1–11, Portland, Oregon, USA, 2006. ACM.

[10] M. Robillard, R. Walker, and T. Zimmermann. Recommendation systems for software engineering. *Software, IEEE*, 27(4):80 –86, july-aug. 2010.

[11] S. Thummalapenta and T. Xie. Parseweb: a programmer assistant for reusing open source code on the web. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 204–213, New York, NY, USA, 2007. ACM.

[12] F. W. Warr and M. P. Robillard. Suade: Topology-based searches for software investigation. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 780–783, Washington, DC, USA, 2007. IEEE Computer Society.

[13] Y. Ye and G. Fischer. Supporting reuse by delivering task-relevant and personalized information. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 513–523, New York, NY, USA, 2002. ACM.

[14] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 563–572, Washington, DC, USA, 2004. IEEE Computer Society.

[15] G. L. Zuniga. Ontology: Its transformation from philosophy to information systems. In *Proceedings of the International Conference on Formal Ontology in Information Systems*, pages 187–197. ACM Press, 2001.