Context Modeling and Context Transition Detection in Software Development

Bruno Antunes, Joel Cordeiro and Paulo Gomes
Centre for Informatics and Systems, University of Coimbra, Coimbra, Portugal

Keywords: Context Modeling, Context Capture, Software Development, IDE.

Abstract: As software development projects increase in size and complexity, developers are becoming overloaded and need to cope with a growing amount of contextual information. This information can be captured and processed in order to improve some of the tasks performed by developers during their work. We propose a context model that represents the focus of attention of the developer at each moment. This context model adapts to changes in the focus of attention of the developer through the automatic detection of context transitions. We have developed a prototype that was submitted to an experiment with a group of developers, to collect statistical information about the context modeling process and to manually validate the context transition mechanism.

1 INTRODUCTION

The interest in the many roles of context comes from different fields such as literature, philosophy, linguistics and computer science, with each field proposing its own view of context (Mostefaoui et al., 2004). The term context typically refers to the set of circumstances and facts that surround the center of interest, providing additional information and increasing understanding. The context-aware computing concept was first introduced by Schilit and Theimer (Schilit and Theimer, 1994), where they refer to context as *"location of use, the collection of nearby people and objects, as well as the changes to those objects over time"*. In a similar way, Brown et al. (Brown et al., 1997) define context as location, identities of the people around the user, the time of day, season, temperature, etc. In a more generic definition, Dey and Abowd (Dey and Abowd, 2000) define context as *"any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves"*.

With the increasing dimension of software systems, software development projects have grown in complexity and size, as well as in the number of features and technologies involved. During their work, software developers need to cope with a large amount of contextual information that is typically not captured and processed in order to enrich their work environment. Especially, in the IDE, developers deal with dozens of different artifacts at the same time. The software development process requires that developers repeatedly switch between different artifacts, which often depends on searching for these artifacts in the source code structure. The workspace of developers frequently comprises hundreds, if not thousands, of artifacts, which makes the task of searching for relevant artifacts very time consuming, especially when repeated very often. The contextual information associated to the work of a developer can be used to identify the source code artifacts that are more relevant at a specific point in time. Although the work of a developer is typically task oriented, it is too complex and dynamic to be easily sliced into simple tasks. The developer often addresses more than one task in a short period of time, or even at the same time, and the switches between these tasks are not explicit. This behaviour makes it difficult to identify the context of a task and to know when the developer switches between tasks. We believe that more important than identifying the context for each one of the tasks the developer has at hands, is to understand what is the focus of attention of the developer at each moment and adapt as it changes.

We propose a context model that represents the focus of attention of the developer at each moment. This context model comprises a structural and a lexical dimensions. The structural context focus on the source code artifacts and their structural relations, while the lexical context focus on the terms used to represent these artifacts. This context model adapts to changes in the focus of attention of the developer, automatically detecting context transitions, either to a new context or a previous one. We have implemented

Antunes B., Cordeiro J. and Gomes P..
Context Modeling and Context Transition Detection in Software Development.
DOI: 10.5220/0004084204770484
In *Proceedings of the 7th International Conference on Software Paradigm Trends* (ICSOFT-2012), pages 477-484
ISBN: 978-989-8565-19-8
Copyright © 2012 SCITEPRESS (Science and Technology Publications, Lda.)

a prototype, named SDiC[1] (Software Development in Context), in the form of a plugin that integrates the context modeling and transition mechanisms in Eclipse[2]. The prototype was submitted to an experiment with a group of developers, in order to collect statistical information about the context modeling process and to manually validate the context transition mechanism.

The remaining of the paper starts with an overview of related work. Then we introduce the developer context model and section 4 explains how the context transition is processed. The prototype developed is described in section 5. The experimentation and results discussion is presented in section 6. Finally, section 7 concludes the work with some final remarks and future work.

2 RELATED WORK

Kersten and Murphy (Kersten and Murphy, 2006) have been working on a model for representing tasks and their context. The task context is derived from an interaction history that comprises a sequence of interaction events, representing operations performed on a software program's artifact. They then use the information in a task context either to help focus the information displayed in the IDE, or to automate the retrieval of relevant information for completing a task. We do not attach the context of the developer to tasks, we see the context model as a continuous and dynamic structure that adapts to the behaviour of the developer. Instead of requiring developers to explicitly define where a task starts and ends, our approach automatically adapts to the changes in the focus of attention of the developer, identifying which artifacts are more relevant for the activities of the developer in each moment.

In the same line of task management and recovery, Parnin and Gorg (Parnin and Gorg, 2006) propose an approach for capturing the context relevant for a task from a programmer's interactions with an IDE, which is then used to aid the programmer recovering the mental state associated with a task and to facilitate the exploration of source code using recommendation systems. The work is essentially focused on methods, while our approach also covers classes and interfaces, and they do not take into consideration the structural relations that exist between these elements. They interpret moves between methods as transitions, differing from our broader interpretation of a context transition, which represents a change in the focus of attention of the developer.

With the belief that customized information retrieval facilities can be used to support the reuse of software components, Henrich and Morgenroth (Henrich and Morgenroth, 2003) propose a framework that enables the search for potentially useful artifacts during software development. Their approach exploits both the relationships between the artifacts and the working context of the developer. The context information is used to refine the search for similar artifacts, as well as to trigger the search process itself. Only a small part of the interaction of the developer with the IDE is considered in the context model, while our approach is fully based on the source code artifacts manipulated by the developer in the IDE.

Holmes and Murphy (Holmes and Murphy, 2005) proposes Strathcona, an Eclipse plugin that allows to search for source code examples. The process consists of extracting the structural context of the code on which a developer is working, when the developer requests for examples. The search is based in different heuristics, such as inheritance relations and method calls, by matching the structural context with the code in the repository. The contextual information is explicitly provided by the developer when searching for source code examples, while our context model is automatically built from the interactions of the developer in the IDE.

Ye and Fischer in (Ye and Fischer, 2002), propose a process called information delivery, which consists in proactively suggesting useful software engineer's needs for components. The process is performed by running continuously as a background process in Emacs, extracting reuse queries by monitoring development activities. They use the JavaDoc comments as context to create a query for retrieving relevant components, while our approach makes use of context to improve the ranking of search results. With the idea that code fragments using similar terms in the identifiers also use similar methods, Heinemann and Hummel (Heinemann and Hummel, 2011) proposes the use of the knowledge embodied in the identifiers as a basis for recommendation of methods. They use the identifiers of a few source lines preceding a method call as context. These approaches focus on the identifiers and comments used in the source code to complement a query used to retrieve relevant artifacts. They do not take into consideration the focus of attention of the developer or the relative relevance of a specific artifact manipulated by the developer.

Warr and Robillard (Warr and Robillard, 2007) developed a plugin for the Eclipse IDE to provide program navigation, by suggesting potentially relevant

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

[2] http://eclipse.org

elements for the current context. The context is created by the user by dragging and dropping elements of interest into a view. It retrieves and ranks all the elements in the project's source code by taking into account their structural relations to any of the elements in the context. The contextual information is provided by the developer through a set of elements of interest, while our approach automatically identifies these elements of interest and how their interest evolves over time.

3 CONTEXT MODELING

The context model we have defined aims to represent the context of the developer in relation to the source code artifacts that are more relevant to the work of the developer at each moment. The model is built from the interactions of the developer with the source code artifacts and evolves over time, as the focus of attention of the developer changes. It comprises a structural and a lexical dimensions, which are described in more detail on the following sections (see figure 1).

3.1 Structural Context

The structural context focus on the source code artifacts and structural relations that are more relevant for the developer in a specific moment. It comprises a list of structural elements and relations with an associated DOI (Degree of Interest), a concept introduced in (Kersten and Murphy, 2006), that represents their relevance in the context of the developer. Next, we describe the structural context in more detail, including the structural elements and structural relations.

3.1.1 Structural Elements

The structural elements represent the source code artifacts with which the developer is interacting. Associated to each artifact is a DOI that is derived from the analysis of the interactions of the developer with that artifact. When the developer opens, closes, activates or edits an artifact, that artifact is considered relevant for the structural context. The DOI of an artifact changes according to the different interactions that affect that artifact. The impact of each interaction in the DOI of an artifact has been defined based on our experience and some empirical tests. When the artifact is opened, it is added to the structural context and its DOI is increased by 0.4. When the artifact gains focus or is edited, its DOI is increased by 0.2 and 0.1, respectively. When an artifact is closed, its DOI is decreased by -0.4.

As time passes, the DOI of the artifacts is decayed, so that the relevance of an artifact to the context of the developer decreases if it is not used over time. The decay is executed every five minutes. To prevent loosing the context if the developer is distracted or away for some reason, the decay is only executed if the developer has been active during that time interval. When the DOI of an artifact reaches zero, the artifact is removed from the structural context. The DOI of each artifact is represented in the interval $[0,1]$, by normalizing the original value using (1).

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

As shown in figure 1, the structural elements *ContextModel*, *ContextElement* and *ContextElement.setDOI(int)* were added to the structural context because they were manipulated by the developer at some point in time. The relevance of these elements to the developer is given by their DOI values, which evolved according to the different interactions of the developer with that elements over time.

3.1.2 Structural Relations

The structural relations represent the relevance of the relations that exist between the source code artifacts that are manipulated by the developer. These relations represent the structural relations that exist in the Java programming language, but are common to most of the object oriented programming languages, and can be categorized in three groups: inheritance (*extensionOf* and *implementationOf*), composition (*attributeOf* and *methodOf*) and behaviour (*calledBy*, *usedBy*, *parameterOf* and *returnOf*). The relevance of the structural relations can be used to measure the relevance of source code artifacts that are not being used by the developer, but are structurally related with the artifacts that are in the context model.

Because the structural relations are not directly affected by the interactions of the developer, their relevance is derived from the structural elements that exist in the structural context. When two, or more, structural elements are bound by one of these relations, that relation is added to the structural context. Associated with each relation is a DOI that represents the relevance of that relation in the context of the developer. The DOI of a relation is computed as an average of the DOI of all structural elements that are bound by that relation. The DOI of a structural relations is updated according to the changes in the elements that are bound by that relation. When no more structural elements are bound by a relation, it is removed from the structural context.

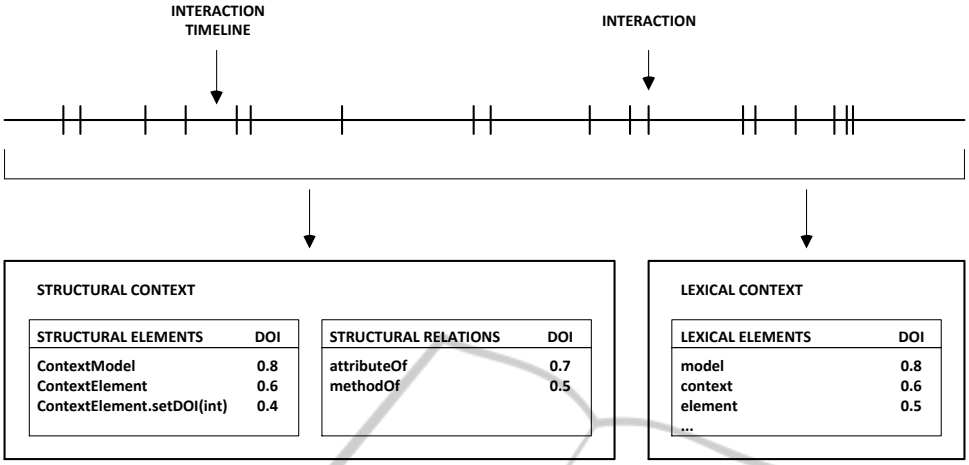

Figure 1: A representation of the context capture and modeling processes.

The structural relations represented in figure 1, *attributeOf* and *methodOf*, were added to the structural context because there are structural elements bound by these relations. The structural element *ContextElement* is an attribute of the structural element *ContextModel*, and *ContextElement.setDOI(int)* is a method of *ContextElement*. The DOI associated with these relations represents the average DOI of the elements that are bound by them.

3.2 Lexical Context

The lexical context focus on the terms that are more relevant in the context of the developer. It comprises a list of terms that are extracted from the names of the source code artifacts that are manipulated by the developer. The name of source code artifacts in the Java programming language typically follow the Camel Case[3] naming standard, resulting in one or more terms joined without spaces and with the first letter of each element capitalized. We use this characteristic to extract the different terms associated with an artifact. The lexical context comprises all the terms extracted from the structural elements in the structural context. Similarly to the elements and relations in the structural context, the relevance of each term is given by its DOI. The DOI of a term is computed as an average of the DOI of the structural elements from which the term was extracted. When the structural elements change, the lexical context is updated accordingly.

Because terms are not explicitly related as the source code artifacts, the terms used in the names of all the source code artifacts that exist in the workspace of the developer are analyzed in order to identify terms that are related by co-occurrence. The co-occurrence relation is created when two terms are found in the name of an artifact. In a linguistic sense, co-occurrence can be interpreted as an indicator of semantic proximity (Harris, 1954). We use co-occurrence as a measure of proximity between two terms, assuming that if the terms are used together to represent the same entity, that means they are related. These relations are then used to understand how the source code artifacts are related with each other from a lexical point of view. We also identify terms that are very frequent, such as *get*, *set*, etc. These very frequent terms co-occur with a variety of other terms and end up connecting almost every term in a distance of three relations. To avoid this, we chose to ignore all the terms that would fall in the top 30% of all term occurrences.

The terms in the lexical context of figure 1, *model*, *context*, *element*, etc, were extracted from the elements in the structural context. Their DOI represents the average DOI of the structural elements referenced by them.

4 CONTEXT TRANSITION

As the focus of attention of the developer changes, the notion of what is relevant to the work of the developer also changes and the context model must be adapted accordingly. The context model described was designed to represent the focus of attention of the developer at each moment, but does not provide, by itself, the mechanisms needed to adapt as the focus of attention changes, which sometimes happens very fast. Because the developer commonly addresses more than one task in a short period of time, or even

[3]http://en.wikipedia.org/wiki/CamelCase

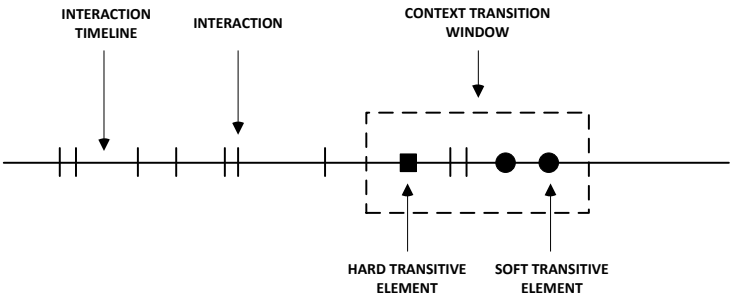

Figure 2: A representation of the context transition window applied to the interaction timeline.

at the same time, the focus of attention is dispersed through different parts of the source code structure. This means that, in fact, more than one context model exist in parallel, and they must be activated and deactivated as the focus of attention changes. This issue has been addressed with a mechanism to deal with context transitions, as the focus of attention of the developer changes. This way we may have several context models, that are stored in a context model pool, with only one context model active at each moment. The system automatically detects the changes in the focus of attention of the developer and decides whether a new context should be created or an existing one should be activated.

To detect changes in the focus of attention of the developer, we rely on the way source code artifacts added to the context model are related with those that are already in the context model. When the attention of the developer shifts to a different part of the source code structure, it is expected to see, in a short period of time, a reasonable number of interactions with source code artifacts that have no relation with those in the current context model. We assume that in such a situation, the system must adapt to the change that is occurring in the behaviour of the developer and make a transition to a new context, or an existing one. To detect these situations we have defined a mechanism based on a fixed time window and a set of transitive elements (see figure 2). The time window is used to represent the time span within which a certain number of elements having no relation with the current context will start a context transition. We call these elements of transitive elements, and they can be either hard or soft transitive. The hard transitive elements are those that have absolutely no relation with the elements in the current context. The soft transitive elements are those that are are related only with hard transitive, or other soft transitive, elements. The time window moves along the interaction timeline as time passes. The hard or soft transitive elements that reach the limit of the time window are no longer marked as transitive elements. When the number of hard transitive elements reaches a threshold of 3, or the number of soft transitive elements reach a threshold of 6, within a context transition window of 3 minutes, a context transition is initiated.

When a context transition is detected, the system must remove the transitive elements from the current context, deactivate it and decide if a new context should be created or an existing one should be activated. To activate an existing context, one must assure that the developer is changing the focus of attention to a part of the source code structure that originated the existing context. The system decides if an existing context should be activated by comparing its elements with the transitive elements, those that were used to detect the context transition in the first place. The similarity between the two sets of elements is computed using the Jaccard index (Jaccard, 1901), also known as the Jaccard similarity coefficient, which is a statistic measure used for comparing the similarity between sample sets. When the similarity between the two sets is greater than a threshold of 0.5, the existing context is activated. If the threshold is not reached, a new context is created, the transitive elements are added to this new context and then it is activated.

5 PROTOTYPE

We have implemented a prototype in the form of a plugin, that integrates the context modeling and context transition mechanisms in Eclipse. The activity of the developer is monitored in the background in order to build the context model and detect context transitions.

The prototype provides an interface where all the information related with the context modeling and context transition processes can be consulted (see figure 3). The interface shows a list of all the context models that have been created (see 1 in figure 3). When one of these context models is selected, all the information related with the selected context model is presented, including information about the struc-

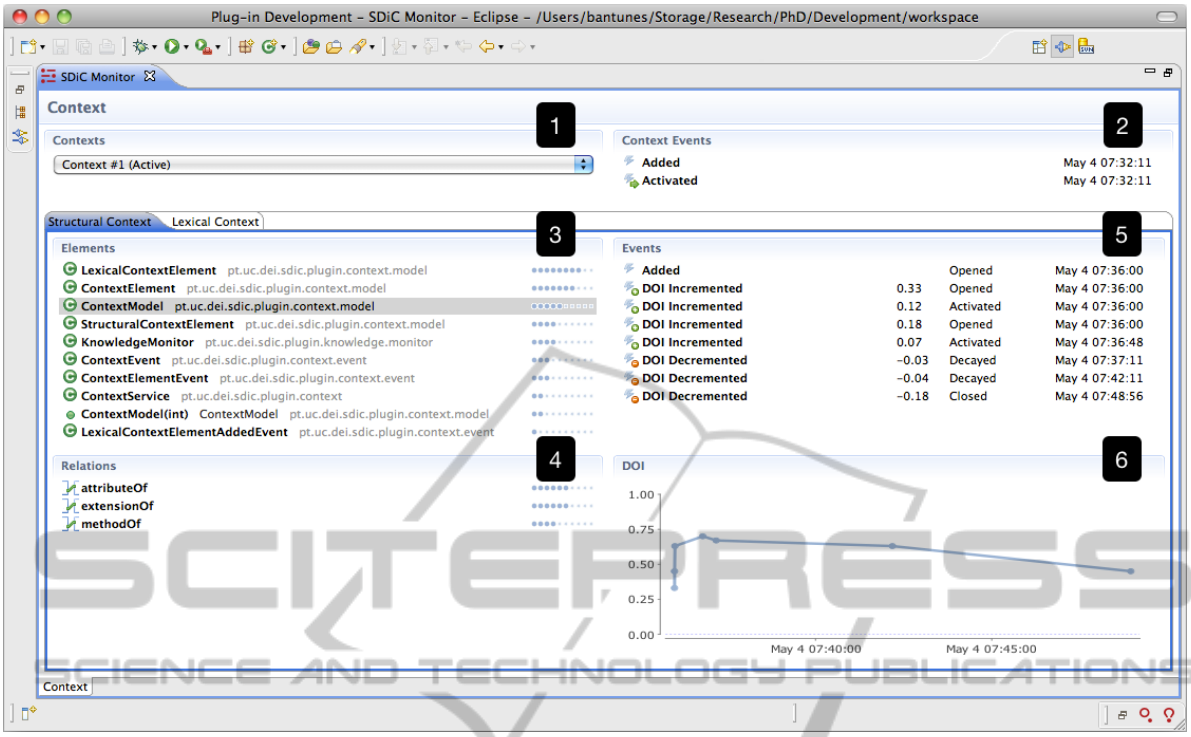

Figure 3: A screenshot of the prototype, showing information about the structural context.

tural and lexical contexts. A list of events (Added, Removed, Activated and Deactivated) associated with the context model is presented (see 2 in figure 3). Concerning the structural context, we can see the structural elements (see 3 in figure 3) and structural relations (see 4 in figure 3) that exist in the selected context model, as well as a visual representation of their current DOI. The hard and soft transitive elements are shown in gray, with the soft transitive elements having a lighter grey than the others. When one of the elements or relations is selected, a list of the events (Added, Removed, DOI Incremented and DOI Decremented) that affected that element are presented (see 5 in figure 3) and a chart representing the evolution of its DOI over time is shown (see 6 in figure 3). Concerning the lexical context model, we can see the terms that exist in the current context model. Similarly to the structural context interface, when a term is selected, the list of events that affected the term is presented and a chart representing the evolution of its DOI over time is shown.

6 EXPERIMENTATION

We have created an experiment to validate our approach in the field, with developers using the prototype during their work. The experiment was conducted during a period of about two weeks, with a group of four developers from different software houses and four students from a computer science graduation, all of them using Eclipse to develop source code in the Java programming language. The objective of the experimentation was to collect statistical data about the context modeling and transition mechanisms, as well as to manually validate the context transitions processed by the system.

Concerning the context modeling mechanism, we have collected statistical data about new elements added to the context model and how they were related with the elements that were already in the context model. This information would allow us to better understand how the source code artifacts manipulated by developers are related with each other and how this could be used to improve the context modeling and transition processes. We have analyzed a total of 9210 elements added to the context model. To evaluate how these elements were related with existing elements, we have verified if they were related within a distance of 3 relations with the top 15 elements with higher DOI of both the structural and the lexical contexts. About 88% of the elements were structurally related with at least one structural element, with an average distance of 2.5 relations, and about 90% were lexically related with at least one lexical element, with an

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

Table 1: Average number of structural and lexical elements.

Average Structural Elements	16.0
Average Structural Related Elements	5.8
Average Structural Unrelated Elements	5.9
Average Lexical Elements	18.4
Average Lexical Related Elements	26.8
Average Lexical Unrelated Elements	2.3

average distance of 2 relations. These numbers show that most of the source code artifacts needed by developers were related with at least one of the artifacts manipulated before, within a distance of about two relations. Based on this, we believe that the use of our context model to rank, elicit and filter relevant source code artifacts for the developer is very promising. Assuming that a source code artifact needed by the developer is likely to be related with the artifacts being manipulated, we can use the proximity between this artifact and the context model to assess its relevance to the developer.

In table 1 we present the average number of structural and lexical elements, as well as the average number of elements related and unrelated with the added element. The lexical related elements have an higher average due to the fact that a source code artifact name typically comprises more than one term, and each match between one of these terms and the terms in the lexical context was considered. The results show that the source code artifacts added to the context model were structurally related with a an average of about 30% of the elements that were already in the context model. The results of the lexical elements are even more expressive. This reinforces the idea that the source code artifacts manipulated by developers are highly correlated. We have also analyzed the types of relations that are more common between the added elements and the existing elements. The percentage of times each relation appeared is shown in table 2. The composition and behavior relations are by far the most common, as expected.

With respect to the context transition process, we have collected statistical information about 55 context transitions, presented in table 3. All the context transitions led to the creation of a new context. We could conclude that a transition to a previous context is something very uncommon, but we have to collect more data in order to understand if improvements can be made in the identification of similar contexts. The use of the Jaccard similarity coefficient between the set of all elements in the existing context and the set of transitive elements may be the cause for this behavior. Because the existing context most of the times contains many more elements than the number of transitive elements, which makes it very difficult to reach the similarity threshold of 0.5. We plan to improve the process by focusing only in the transition elements. Because the transition elements are more relevant for the context transition process than the others, we may assume that if an existing context contains most of the transition elements, then this context model is a good candidate to be activated. Also, we could conclude that context transitions are more often caused by reaching the hard transitive elements threshold than by reaching the soft transitive elements threshold.

Table 2: Percentage of times each relation appeared in the relations between added and existing context elements.

extensionOf	5.5%
implementationOf	1.0%
attributeOf	42.3%
methodOf	90.3%
calledBy	69.7%
usedBy	30.5%
parameterOf	9.3%
returnOf	2.3%

Table 3: Statistical information collected about the context transition process.

Context Transition (New Context)	100%
Context Transition (Existing Context)	0%
Context Transition (Hard Transition)	67%
Context Transition (Soft Transition)	33%

Finally, we asked the developers to evaluate how each context transition detected by the system could be identified as change in their focus of attention. They were presented with the structural elements that were in the context model before the transition and the elements that were used to detect the transition (both hard and soft transitive). They were asked to rate how the context transition would be related with a change in their focus of attention in a scale from 1 (Poorly Related) to 5 (Highly Related). The average score for the 55 context transitions evaluated was 3.0. The average score obtained is not conclusive, but is encouraging, at least. One of the problems we have faced is that developers have some difficulties understanding the concepts of context transition and focus of attention, which can have lead to misjudgment in the evaluation process.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

7 CONCLUSIONS

We have presented an approach to context modeling and context transition detection in software development. The context model combines a structural and a lexical dimensions, to represent the source code artifacts, structural relations and terms that are more relevant for the developer in a specific moment in time. The context transition detection mechanism allows the context model to automatically adapt to the changes in the focus of attention of the developer. We have implemented a prototype that integrates our approach in Eclipse. This prototype was submitted to an experiment with a group of developers to collect statistical information about the context modelling process and to manually validate the context transition mechanism. The statistical information collected shows that the source code artifacts manipulated by the developer are highly correlated, leading us to believe that the use of a context model to assess the relevancy of a source code artifact to the developer is very promising. The human evaluation of the context transition mechanism was not conclusive, but the results are nevertheless encouraging, considering the fact that developers have some difficulties in understanding the concept of context transition.

As future work we plan to improve the context modeling and context transition processes, taking into consideration some of the issues that were identified. Also, we want to evaluate if the lexical context can be used to detect context transitions and how it would impact the the current context transition mechanism. Next, we want to apply the context model developed to improve the context-based retrieval of source code artifacts in the IDE. The context model can be used to rank, elicit and filter source code artifacts based on their proximity to the context model.

ACKNOWLEDGEMENTS

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

REFERENCES

Brown, P. J., Bovey, J. D., and Chen, X. (1997). Context-aware applications: From the laboratory to the marketplace. *Personal Communications, IEEE*, 4:58–64.

Dey, A. K. and Abowd, G. D. (2000). Towards a better understanding of context and context-awareness. In *CHI 2000 Workshop on the What, Who, Where, When, and How of Context-Awareness*, The Hague, The Netherlands.

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

Heinemann, L. and Hummel, B. (2011). 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. ACM.

Henrich, A. and Morgenroth, K. (2003). Supporting collaborative software development by context-aware information retrieval facilities. In *14th International Workshop on Database and Expert Systems Applications, 2003. Proceedings.*, pages 249 – 253.

Holmes, R. and Murphy, G. C. (2005). 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. ACM.

Jaccard, P. (1901). Étude comparative de la distribution florale dans une portion des Alpes et des Jura. *Bulletin del la Société Vaudoise des Sciences Naturelles*, 37:547–579.

Kersten, M. and Murphy, G. C. (2006). 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. ACM.

Mostefaoui, G. K., Pasquier-Rocha, J., and Brezillon, P. (2004). Context-aware computing: A guide for the pervasive computing community. In *Proceedings of the IEEE/ACS International Conference on Pervasive Services, ICPS 2004*, pages 39–48.

Parnin, C. and Gorg, C. (2006). Building usage contexts during program comprehension. In *Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06)*, pages 13–22.

Schilit, B. and Theimer, M. (1994). Disseminating active map information to mobile hosts. *IEEE Network*, pages 22–32.

Warr, F. W. and Robillard, M. P. (2007). 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. IEEE Computer Society.

Ye, Y. and Fischer, G. (2002). 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. ACM.