A context model for IDE-based recommendation systems

Marko Gasparic\textsuperscript{a,\textdagger}, Gail C. Murphy\textsuperscript{b}, Francesco Ricci \textsuperscript{a}

\textsuperscript{a} Free University of Bozen-Bolzano, Dominikanerplatz 3, 39100 Bolzano, Italy
\textsuperscript{b} University of British Columbia, 201–2366 Main Mall, V6T 1Z4 Vancouver BC, Canada

\section*{Article Info}

Article history:
Received 19 September 2015
Revised 30 August 2016
Accepted 10 September 2016
Available online 12 September 2016

Keywords:
Context
Usage
Integrated development environment
Commands
Software development
Recommendation systems for software engineering

\section*{Abstract}

Context, as modeled through variables called contextual factors, can improve human-computer interaction. To date, in applications supporting software development, such as integrated development environments (IDEs) and recommendation systems for software engineering (RSSEs), contextual factors have generally been constrained to project artifacts, such as source code. In this paper, we present a context model that includes thirteen contextual factors, which capture various situations in which developers interact with an IDE. This context model can be used to support and enhance user interaction with an IDE or to improve the accuracy and timing of recommendations produced by RSSEs.

To assess whether the proposed contextual factors are informative for a context model, we statistically evaluated the correlations between IDE command usage and different situations, as they are described by the factors. If a contextual factor correlates with the usage of a command this means that the user is using the command differently when the values of the contextual factor change. We discovered that different factors correlate with different commands and that all the factors correlate with some commands, hence, when a context change is detected, we can also expect a change in the interaction with an IDE.

\section*{1. Introduction}

Professional software developers spend approximately one third of their time working in an integrated development environment (IDE), which makes it the most used application during a working day of a software developer (Sillitti et al., 2012). Modern IDEs, such as Eclipse\textsuperscript{1}, Visual Studio\textsuperscript{2}, and IntelliJ IDEA\textsuperscript{3}, bring together multiple tools to create and manipulate software project artifacts, such as advanced editors to write source code, testing tools that display test suite execution results, automatic compilers that run in the background, and debuggers that show the state of variables and the current set of functions on a call stack, amongst other tools.

Due to the complexity of the work software developers undertake in an IDE, different supporting tools have been developed. One popular and promising group of such tools are recommendation systems for software engineering (RSSEs). They are defined as tools that provide “information items estimated to be valuable for a software engineering task in a given context” (Robillard et al., 2010, pp. 81). RSSEs aim to reduce developers’ cognitive load (e.g., Holmes et al., 2009), reduce the amount of time developers spend looking for a specific information (e.g., Cubranic et al., 2005), help improve the quality of the system under development (e.g., Barbosa et al., 2012), and increase the efficiency and effectiveness of the development process (e.g., Zhang et al., 2012).

To date, the context taken into account by existing RSSEs is largely limited to characteristics of common project artifacts, such as source code and development tasks, and to information about user interaction with these artifacts (Gasparic and Janes, 2016). The need for development of context models that go beyond interaction events and common project artifacts to help achieve higher accuracy of RSSEs or provide additional functionality in an IDE has been expressed by many researchers (e.g., Bavota et al., 2014; Brown et al., 1990; Holmes et al., 2006; Hummel et al., 2007; Janjic et al., 2014, and Kim and Meng, 2014). Our focus is on a context model that describes the situations in which developers use an IDE. We conjecture that this type of information could improve the context awareness of support tools, such as RSSEs, which may lead to more accurate recommendations and improved delivery, particularly, ordering and timing of recommendations. Examples of existing RSSEs that may benefit from knowing the context of developers when they are using an IDE include: Autumn-Leaves (Roethlisberger et al., 2009), which closes irrelevant win-

\textsuperscript{1} http://www.eclipse.org
\textsuperscript{2} http://www.visualstudio.com
\textsuperscript{3} http://www.jetbrains.com/idea/

\textsuperscript{\textdagger} Corresponding author.
\textsuperscript{E-mail addresses: marko.gasparic@stud-inf.unibz.it, m.gasparic@gmail.com (M. Gasparic), murphy@cs.ubc.ca (G.C. Murphy), francesco.ricci@unibz.it (F. Ricci).}

Received 19 September 2015
Revised 30 August 2016
Accepted 10 September 2016
Available online 12 September 2016

http://dx.doi.org/10.1016/j.jss.2016.09.012
0164-1212/© 2016 Elsevier Inc. All rights reserved.
dows or tabs in an editor, and could benefit from taking into account the information about the type of the interaction with particular artifacts, since their importance varies depending on the activity during which they were accessed (Minelli et al., 2014) and Quick Fix Scout (Müşlu et al., 2012), which reorders the initial list of suggested compilation error resolutions, and could adapt the ordering also by taking into account the knowledge of the developer, since, for example, less experienced developers may simply discard too complex suggestions or they may waste a lot of time studying and comparing them.

In this paper, the notion of context is based on Dey's definition:

"Context is 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." (Dey, 2001, pp. 5)

The suggested context model consists of thirteen contextual factors: variables with precise domains of possible values that are used to identify the context. It characterizesdeveloper situations from several perspectives, namely, who, what, when, and where. To demonstrate the potential of the factors for describing the situations in which an IDE is used, we present the results of an evaluation in which we examined the correlations between different contexts—as described by the proposed contextual factors—and the usage of IDE commands, through which a user is interacting with an IDE. If a contextual factor correlates with the usage of a command this means that the user is using the command differently, when the values of the contextual factor change. Our results show that different factors correlate with different commands and that all the factors correlate with some commands.

We gathered data about which commands in an IDE were used by eight Eclipse users working in either academia or a medium-sized company for ten weeks. The correlation analysis is similar to other analysis conducted on other contextual models (e.g., Braunhofer et al., 2015; Braunhofer and Ricci, 2016; Codina et al., 2016; Odic et al., 2012, and Odic et al., 2013). Our study provides evidence that the factors included in our model are informative and may be used for describing different situations in which an IDE is used. We believe that this is a basic and necessary step in the process of incorporating a new context model in a specific application, such as a RSSE, despite the fact that such evaluations are rare in the software engineering domain. The next step for us and for the researchers who want to utilize the results of our work is to build a context-aware application on top of our model, and assess the benefits and drawbacks of the model for a more specific purpose.

This paper makes three contributions:

- it provides a discussion on limited forms of context that are used by existing RSSEs that support the development in an IDE,
- it introduces a context model for capturing the different situations of IDE use that can help tailor IDE support tools, most particularly RSSE tools, and
- it shows that the contextual factors included in the model have statistical correlation with the usage of IDE commands, in particular, when the values of the contextual factors change, different developers—as individuals and as a group—use significantly different commands for interacting with an IDE, according to Pearson's χ² and Fisher–Freeman–Halton tests.

We begin with a review of related work on context used by selected RSSEs and context model evaluation methods (Section 2). We then describe the context model precisely (Section 3) and describe how it may be applied to a specific IDE (Section 4). We present the evaluation of the model (Section 5) and discuss various aspects of the use of a context model for IDEs (Section 6) before summarizing the paper (Section 7).

2. Related work

Contextual factors vary greatly across disciplines. For example, in e-commerce, contextual factors include customer’s interest (Palmisano et al., 2008) or time, schedule, partner, temperature, and weather (Oku et al., 2006). In ubiquitous and mobile systems, contextual factors include location of the user, the identity of people and objects around the user, and changes in these elements (Schilit and Theimer, 1994) or date, season, and temperature (Brown et al., 1997).

In software engineering, the contextual factors used by existing tools have largely been implicit and latent, meaning that in retrospect, we can identify some contextual factors, but these factors are not explicitly defined as being contextual by their authors nor is the usage of context motivated in the original work describing these RSSEs. In most cases, the authors of the scientific contributions present only raw data input and the algorithm used for generating recommendations. Generally, the input is limited to project artifacts, such as source code and (manually defined) issues and tasks, and their metadata. In a few cases, the input data includes navigation patterns describing how program structure is being navigated. For a detailed overview of what input types are used by the existing RSSEs, we refer the reader to the systematic literature review conducted by Gasparic and Janes (2016).

In this paper, we posit that a context model should be broader to be able to support and enhance several important applications related to developers’ interactions with IDEs. It is also beneficial if the model is explicitly evaluated before it is introduced in a specific tool, for example, by correlation analysis, which we present in this work as well. In this section, we discuss some relevant RSSE examples (Section 2.1) and we describe earlier work in context model evaluation (Section 2.2).

2.1. Context models in software engineering

In the following sections, we discuss how certain IDE plugins extract context from raw input data and use it to adapt their own behavior or the user interface of an entire IDE.

2.1.1. Improving the IDE command usage

Research shows that software developers use only a small set of features provided in their IDEs. Murphy-Hill (2012) discovered that an average Eclipse user uses 42 different commands out of over 1100 typically available commands, even though the execution of a command, which usually requires pressing a button or a key sequence, is often faster than a manual step-by-step process. There have been a few attempts to recommend commands to software developers to help strive for a more effective use of an IDE.

SpyGlass (Viriyakattiyaporn and Murphy, 2010) is a tool that can recommend three navigation commands. It generates recommendations when it detects inefficient behavior, based on the observations of developer’s navigation over source code artifacts. The contextual factor used by SpyGlass may be considered as recent navigation patterns.

Murphy-Hill et al. (2012) and Zolaktaf and Murphy (2015) describe the approaches that enable a recommendation of any IDE command. They take IDE command usage history as the input and extract command discovery patterns of the users and

---

4 An interested reader may also want to read (Maaiej et al., 2014), which contains a presentation of three exemplary interaction-based RSSEs and a conceptual framework for collecting and processing interaction data for the purpose of recommendation.
co-occurrence of command executions in the session, to generate personalized command recommendations. The offline evaluations show that taking into account command discovery and co-occurrence improves the accuracy of predicting which command will be discovered next, compared to the standard collaborative-filtering algorithms and “most popular” or “most used” non-personalized suggestions. The state-of-the-art approach, which is presented in (Zolaktaf and Murphy, 2015), takes into account three contextual factors: discovery patterns, recent command usage, and elapsed time since last activity.

2.1.2. Adapting the user interface of an IDE

Developers often have to spend a substantial amount of time to find the information and project artifacts they need to complete a particular task. Since the tasks are rarely limited to one single module and developers often switch between tasks, finding the relevant artifacts on which to work reduces their productivity (Kersten and Murphy, 2006). Moreover, if the workspace is not cleaned up regularly, it leads to the presence of too many opened windows or tabs, which also negatively impact developers’ performance, since they have to spend more and more time locating the window of interest and have to keep a larger mental map of the content and purpose of each opened window (Roethlisberger et al., 2009). In these cases, an IDE can leverage context to help determine which tools and artifacts are used in certain situations within the IDE to help adapt the user interface to the actions being performed by the developer (Brown et al., 1990).

Mylyn (Kersten and Murphy, 2006) is an Eclipse plugin that makes certain artifacts more visible and removes others from the workspace, based on the active task, structural relationships of program artifacts, and developer’s interactions with them during the task execution. The active task must be defined and activated manually. According to Kersten and Murphy (2006), the context is a graph of elements and relationships of program artifacts. In this paper, we also consider the task to be part of the context.

AutumnLeaves (Roethlisberger et al., 2009) is a tool that closes irrelevant windows or tabs in an editor. It takes into account developer’s interaction with opened project artifacts and the similarity of their content to assign weights to the user interface elements visualizing the artifacts. It compares the weights to detect outliers that the developer will unlikely revisit. In the model proposed in this paper, we identified one contextual factor that can help to tame the above mentioned problem: relevance of opened windows or tabs.

Several other tools try to improve developer’s awareness of the existing project artifacts or try to support the navigation over them. However, instead of automatically manipulating the visualization of the artifacts opened in an IDE, they only show a list of recommended artifacts. They are, like Mylyn and AutumnLeaves, extracting the relevance measures of project artifacts by performing source code (e.g., Robillard, 2005; Shimada et al., 2009 and Warr and Robillard, 2007) or navigation logs analyses (e.g., DeLine et al., 2005 and Singer et al., 2005).

2.1.3. Adapting the order of recommendations

To accelerate developer’s work, modern IDEs provide tools that can reduce cognitive load and shorten the time needed for typing. Two widely used tools are “auto-complete”, which provides a list of words, such as variable names, class names, and method calls, based on pattern matching of the typed text, and “quick fix”, which recommends automatic source code transformations for compilation errors (Murphy et al., 2006).

OCompletion (Robbés and Lanza, 2010) is an RSSE that generates shorter and better-ordered list of auto-complete suggestions than the standard auto-completion tools, which often provide long and alphabetically-ordered lists of suggestions. The best performing algorithm implemented in OCompletion analyzes previous editing events to form a vocabulary of possible completions and it takes into account (the beginning of) the typed word and the reference type of the receiver, such as the class of a variable, for example. In the model proposed in this paper, we identified three contextual factors that can help to tame the above mentioned problem: vocabulary, typed word, and reference type of the receiver.

Quick Fix Scout (Muslu et al., 2012) reorders the initial list of suggested compilation error resolutions. It calculates the consequences of the execution of each individual suggestion and promotes those that resolve the largest number of errors. In the model proposed in this paper, we identified two contextual factors that can help to tame the above mentioned problem: suggested compilation error resolutions and number of resolved errors.

2.1.4. Providing relevant examples

Context has also been used in fields that are not directly related to interaction with an IDE. As examples of such tools, we briefly discuss two representative RSSEs for recommending source code examples.

Strathcona (Holmes and Murphy, 2005) is an RSSE that analyzes an indicated source code fragment, creates a query from the extracted information, and uses it to obtain relevant source code examples, which are then visualized in Eclipse. In this system, the context is represented by the source code fragment query.

Code Conjurer (Hummel et al., 2008) is similar to Strathcona, but it does not require a manual selection of the relevant source code; instead, it forms a query based on the analysis of the artifacts under development. Code Conjurer can find reusable source code for a specific JUnit test, and it can adapt the final ranking of the recommendations by taking into account additional metrics, such as lines of code and cyclomatic complexity (Janjic et al., 2014). In the model proposed in this paper, we identified two contextual factors: artifacts under development query and type of the artifact under development.

2.1.5. Improving the timing of recommendations

Proactive RSSEs generate and present recommendations automatically, without waiting for an explicit request made by the user. They have a large potential to help developers when they are in need, but detecting the right moment for pushing recommendations is not easy to automatically compute. Task engagement can be used as a good indicator of human interruptibility, which is a key condition to detect the time interval for the delivery of recommendations that will be accepted by the user (Fogarty et al., 2005).

Fogarty et al. (2005) observed how Eclipse users interact with the IDE, by monitoring reading, coding, code navigation, interface navigation, task switching, and debugging activities. They discovered that the developers were less interruptible when they performed certain editing events. We could characterize this type of information as the developer current status contextual factor, which has two alternative values: “interruptible” and “uninterruptible”.

Carter and Dewan (2010) built a tool that can detect if developers are having difficulty, and when it detects that, it updates developers’ public status, so that collaborators can see that they are stuck. The approach is similar to (Fogarty et al., 2005): the tool observes the interactions with an IDE and extracts “having difficulty” or “not having difficulty” values of the developer current status contextual factor.

2.1.6. Comparison of the existing context models with the suggested context model

Our model contains thirteen contextual factors, which we grouped into four categories:
Table 1
Summary and mapping of the contextual factors used in the existing RSSEs and in our model.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Developer current status</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Developer general experience</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Developer IDE experience</td>
<td>×</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Developer unique identifier</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Artifacts under development query</td>
<td>×</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Complexity of the artifact under development</td>
<td>×</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Current activity</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Discovery patterns</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Graph of elements and relationships of program artifacts</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Length of the artifact under development</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of resolved errors</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Previous activity</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Recent command usage</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Recent navigation patterns</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reference type of the receiver</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Relevance of opened windows or tabs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Source code fragment query</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Suggested compilation error resolutions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Task</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type of the artifact under development</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Typed word</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vocabulary</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Day of the week</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Elapsed time since last activity</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time of the day</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Active user interface elements</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IDE instance</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>User interface element with focus</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **who** — to capture the information about the developer who is interacting with an IDE;
- **what** — to capture the information about what developers are doing with an IDE and which project artifacts they are using;
- **when** — to capture the information about temporal aspects of developer's interaction with an IDE; and
- **where** — to capture the information about the IDE environment with which a developer is interacting.

In Table 1, we list all the factors included in our model, as they are grouped in the categories, together with the mapping of the context used by the existing tools mentioned in the previous sections. We discuss the suggested factors in detail in Section 3. To the best of our knowledge, the model presented in this paper is the first context model that has been designed particularly for characterizing different situations in which a developer uses an IDE. It includes more contextual factors than the existing models and characterizes the situations from several perspectives. In Section 6, we comment on how the contextual factors included in our model could lead to an easier interaction with an IDE, improve the accuracy of the recommendations provided by different RSSEs, or improve the timing of the recommendation delivery.

### 2.2. Evaluating a context model

In information theory, the goal of feature selection methods is to eliminate the features that do not provide meaningful information (Koller and Sahami, 1995). In this paper, we want to assess which contextual factors included in the proposed model provide meaningful information. Direct evaluation of the context model is important, as irrelevant contextual factors can result in unnecessary data collection and may negatively affect the performance of an entire system built on top of the model (Odic et al., 2012). In this section, we discuss some existing methods for evaluating context models.

Feature selection methods are usually divided into two groups, namely, wrapper and filter methods (Koller and Sahami, 1995). In wrapper methods, the subset of features—contextual factors, in our case—is selected based on the performance of the algorithm or the system built on top of the model. This type of evaluation is expensive and requires an implementation of a particular algorithm or a system before the evaluation of the context model is performed. We are aware of four studies in the software engineering domain in which context models were evaluated with a wrapper method. Piorkowski et al. (2011) measured the accuracy of how
well different navigation models predict which artifact is going to be accessed next. Murphy-Hill et al. (2012) and Zolaktaf and Murphy (2015) measured how accurately different algorithms, based on different models, predict which of the IDE commands that were not yet used was executed first. And Murphy-Hill et al. (2012) evaluated how many useful and novel IDE command recommendations differ when compared to implemented information.

In filter methods, the context model is evaluated directly, without integration in an algorithm or a tool, by using a proxy measure. We are not aware of any such evaluation of a context model in software engineering, but, we are aware of several such evaluations of the models devised for recommender systems in other fields.

As one example, Baltrunas et al. (2012) devised a method to evaluate the relevance of contextual factors of a given context model, where users are asked to judge whether a contextual factor influences the rating given a certain value of the factor. They implemented the evaluated model in a mobile recommender system.

As the second example, Odic et al. (2013) devised a statistical framework for the evaluation of contextual factors’ relevance. Additionally, using a wrapper method, they showed that the factor selection based on statistical tests out-performs survey-based approaches in terms of rating predictions accuracy in a movie recommender system. In other words, the selection of the set of factors based on the statistical evaluation enables more precise selection, which, when used in a particular context-aware recommender system, leads to better performance of the system. Using statistical tests in feature selection is not uncommon (Forman, 2003; Liu and Motoda, 1998; Manning et al., 2008). Statistical tests included in the framework suggested by Odic et al. (2013) were used also in context model evaluations performed by other researchers, e.g., (Braunhofer et al., 2015), (Braunhofer and Ricci, 2016), and (Codina et al., 2016).

To evaluate the context model proposed in this paper, we used a filter method, as the model is not specific to one application. We adopted the approach suggested by Odic et al. (2013) and evaluated the relevance of each contextual factor individually, using Pearson’s χ² and Fisher–Freeman–Halton statistical tests.

3. Context model

A context model is a set of contextual factors that characterize a situation. As the set of all contextual factors needed to represent a situation can be excessively large (Dey, 2010) and can contain factors that are difficult to acquire (Maalej et al., 2014), or even factors that are not informative (Odic et al., 2012), we chose to limit the contextual factors to those that have an impact on how a developer works with an IDE. We further consider the following constraints:

• data acquisition should not require repetitive or substantial manual effort,
• contextual factors should be general enough to allow to compare the situations of different developers, also outside organization, and
• when a value of a contextual factor changes, the developer’s interaction with an IDE should change as well, i.e., the contextual factor should be informative.

Explicitly requesting the user to specify the current value of a contextual factor may not scale well in certain situations or the accuracy of the input may be questionable. For instance, asking a developer to provide the level of expertise for each part of the product she opens or each activity she performs in an IDE would impose a great burden; moreover, she may not have the ability to evaluate her own expertise on such a fine-grained level (Ying and Robillard, 2014). On the other hand, developers may simply refuse to invest additional effort to use the system or they forget to do the required manual work (Maalej et al., 2014). In (Kersten and Murphy, 2006), for example, the user may forget to activate a new task, when she starts working on it. Since we cannot assume that developers are willing to change their working process or invest a significant effort to use a new system, we decided to focus only on factors for which the raw data can be collected automatically or where the manual input would require very low user effort, such as stating how many years of experience a developer has, based on which we estimate the value of developer general knowledge contextual factor.

In general, the granularity of contextual factors can vary: we can observe keystrokes or the characteristics of an abstract task that the developer is working on (Maalej et al., 2014). While certain authors claim that context-aware systems should access as much information about the development situation as possible (e.g., Janjic et al., 2014), there is a trade-off between the ease of extracting context and its generalizability on one hand and the precision of the system on the other (Inozemtseva et al., 2014). According to Robillard and Walker (2014), the authors of RSSEs have to accept that the context will generally be incomplete and noisy, since the more precise the information is, the more accurate the recommendations can be, but the less likely the user will need them. For instance, if we observe a developer using an IDE while modifying an entity class, taking into account the information about the exact identifiers of project artifacts will hamper the comparison of the situations, since developers outside the organization will always be in a different situation, as described by that factor. In our work, we try to build a context model that can be used to characterize situations of developers using an IDE and enables meaningful comparisons of the situations of a single developer, as well as the situations of any two different developers.

Since there are many IDEs available on the market, we cannot build a model that is directly applicable to all of them. Instead, we have created a model that is independent from the IDE, and is presented in this section, while in the next section we show how it can be applied in practice to a specific IDE. The inclusion of factors is based on the previous literature and our own experience. To confirm our hypotheses to include certain contextual factors, we performed a case study and examined statistical correlations of factors with IDE command usage, which is one of the exclusion criteria. As can be seen in Section 5, all the considered factors are informative in describing in which situations different commands are used and are, consequently, included in the model. As we have already briefly mentioned, our model consists of thirteen contextual factors, which are listed in Table 2. According to Dey’s template (Dey, 2010), we group the contextual factors into four categories: who (Section 3.1), what (Section 3.2), when (Section 3.3), and where (Section 3.4). Possible values of these factors, i.e., contextual factor domains, can be used to describe what a developer is doing at a particular point in time and how the work a developer is performing is being conducted within an IDE. We describe each category of factors in turn.

3.1. Who

The contextual factors in the who category capture general information about the developer who is interacting with an IDE. Ac-

Using the intuition is a common approach in context-aware system design (Inozemtseva et al., 2014).

We only use the categorization to ease the understanding of the paper. The suggested factors could be grouped also in a different way or not grouped at all.

For some of the contextual factors, the domain depends on the IDE in use and the participants. We mark with “a” all of the domains which include only examples of possible values as opposed to a list of predefined and definite contextual factor values.
Table 2

<table>
<thead>
<tr>
<th>Category</th>
<th>Contextual factor</th>
<th>Contextual factor domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Who</td>
<td>Developer unique identifier</td>
<td>Developer A, Developer B, ...</td>
</tr>
<tr>
<td></td>
<td>Developer general experience</td>
<td>novice, competent, expert</td>
</tr>
<tr>
<td></td>
<td>Developer IDE experience</td>
<td>novice, expert</td>
</tr>
<tr>
<td>What</td>
<td>Current activity</td>
<td>reading, editing, navigating, debugging, using version control, reviewing code, other</td>
</tr>
<tr>
<td></td>
<td>Previous activity</td>
<td>reading, editing, navigating, debugging, using version control, reviewing code, break, other</td>
</tr>
<tr>
<td></td>
<td>Type of the artifact under development</td>
<td>JavaScript file, XML file, Java class, Java method, ...</td>
</tr>
<tr>
<td></td>
<td>Length of the artifact under development</td>
<td>short, medium, long</td>
</tr>
<tr>
<td></td>
<td>Complexity of the artifact under development</td>
<td>simple, medium, complex</td>
</tr>
<tr>
<td>When</td>
<td>Time of the day</td>
<td>morning, afternoon, evening, night</td>
</tr>
<tr>
<td></td>
<td>Day of the week</td>
<td>Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday</td>
</tr>
<tr>
<td>Where</td>
<td>IDE instance</td>
<td>Eclipse Luna, Subclipse, oXygen, ...</td>
</tr>
<tr>
<td></td>
<td>Active user interface elements</td>
<td>[Java editor, XML editor, Project Explorer view], [Java editor, Console view, Outline view, Project Explorer view], ...</td>
</tr>
<tr>
<td></td>
<td>User interface element with focus</td>
<td>[Java editor, XML editor, Project Explorer, ...]</td>
</tr>
</tbody>
</table>

* actual domain depends on the collected data and cannot be predefined.

According to Dey (2010), context is all about the whole situation relevant to an application and its set of users. Thus, primarily, we have to know who are the users, if we want to be able to relate different situations to them. In our model, the developer unique identifier factor refers to each individual IDE user. Potentially, this factor could be replaced by a specific IDE installation or workspace instance, as can be seen also in the model evaluation results presented in Section 5; however, since some developers use different IDE installations and workspace instances, and some installations and workspace instances are used by different developers, information about a developer is more informative.

In knowledge-related activities, experience plays a very important role, even though the level of experience required to use a method or a tool is rarely explicitly defined (Robillard, 1999). In software development, developers' knowledge usually refers to the knowledge developers have about the system they are working on and their general software development experience (Ying and Robillard, 2014). Specialized experience has the greatest impact on the productivity, while the effect of general experience on productivity varies across different tasks (Fong Boh et al., 2007).

According to Fischer (2001), for each user of a high-functionality application, such as IDE, there are well known concepts, vaguely known concepts, and unknown concepts; and vaguely known concepts can be learned faster and easier than unknown. We believe that higher general and specific IDE experience increase the size of vaguely known and well known concepts, respectively; the level of experience affects the IDE usage, and, consequently, may represent a relevant context. In our model, we use developer general experience and developer IDE experience contextual factors to characterize the level and the type of experience of each user.

The developer general experience factor describes the level of overall software development experience of the user. We adopted the possible contextual factor values from Sillito et al. (2008): developers with five or more years of professional experience are classified as experts, developers that have between two and five years of professional experience are classified as competent, and developers with less than two years of professional development experience are classified as novices.

In the developer IDE experience factor, we distinguish between experts and novices, similarly as Murphy-Hill et al. (2012), who analyzed the acceptance of recommended IDE commands and detected significant differences between these two groups. We propose to use the average number of different IDE commands used by an average IDE user as a threshold. Since it is IDE specific, we discuss it further in the next section.

3.2. What

The contextual factors in the what category capture information about what a developer is doing with an IDE and which project artifacts are used during or affected by her actions. Activities refer to a set of certain interaction events (Maelej et al., 2014); in our case, the current activity factor describes what the developer is currently doing within an IDE. We based the values for this factor on Meyer et al. (2014) characterization of developers’ work that is based on observations of professional developers. Meyer et al. describe six primary activity categories with related sub-categories: coding (reading, editing, and navigating), debugging, using version control (reading, accepting, submitting changes), testing performed outside the IDE, reviewing code, and other (Meyer et al., 2014). We eliminate the “testing performed outside the IDE” activity as we are focused on a developer’s usage of an IDE. We also do not limit reading, editing, and navigation activities only to the source code artifacts, but consider the activities performed on any kind of artifact accessed inside an IDE. As “other” we classify all the detected IDE interactions that are not mapped to any other activity.

Prior to whatever activity a developer is currently undertaking, a developer may have undertaken a different activity or may have been idle. We capture this information with the previous activity contextual factor. The possible values of the previous activity are those that can be assigned to the current activity factor and “break”.9 As previous research has shown, the interruptions affect the behavior of developers and as resumption activities differ from regular development activities (Parnin and Görg, 2006; Parnin and Rugaber, 2009), we include the “break” value in the previous activity to better describe whether a developer has continuously worked in an IDE or not, and thus has to regain the previous mental state.

The artifact under development, i.e., the object of the performed activity, has long been recognized as a relevant context in the software engineering community (DeLine et al., 2005; Holmes and Murphy, 2005; Robillard, 2005; Shimada et al., 2009; Warr and Robillard, 2007). The artifacts can be represented at different levels of granularity, for instance, as files as by Singer et al. (2005) or Java elements (e.g., classes, methods, fields, methods calls) as by Kersten and Murphy (2006). Furthermore, we can use exact identifiers of source code elements, such as absolute paths and (unique) identifiers, or abstract descriptions, such as complexity measures.

---

9 Breaks are not included in the current activity, because developers do not interact with an IDE during the break.
For our context model, we chose abstract representation, as it helps to manage the size of the context domain and allows generalizability of the situations. The contextual factors—type of the artifact under development, length of the artifact under development, and complexity of the artifact under development—are used to provide an abstract representation of an artifact. For the last two factors we use a simple three-point scale. Since the domain of type of the artifact under development and the thresholds of contextual factor domains of length of the artifact under development and complexity of the artifact under development depend on the tools used to collect the raw data from which the context values are extracted, we continue the discussion in Section 4.

3.3. When

The contextual factors in the when category capture temporal aspects of developer's interaction with an IDE. We use two factors: time of the day and day of the week, which describe when the work is being performed. We include these factors in the model as there is evidence that on certain days in the week and at certain times of the day, the quality of the source code varies, i.e., buginess of source code commits changes during the week and during the day (Eyolfson et al., 2011; Ghezzi and Gall, 2013; Śliwerski et al., 2005). Thus, developers must be doing something differently in different time intervals.

We split one day into four parts: morning (between 6am and noon), afternoon (between noon and 6pm), evening (between 6pm and midnight), and night (between midnight and 6am). We use basic, seven days scale to specify the days of the week.

3.4. Where

The contextual factors in the where category capture information about the environment with which a developer is interacting. We include factors that describe which parts of the IDE are available and which ones the developer is using.

IDE instance captures what functionality is available to be used. Since most IDEs have a plugin architecture that enables new functionality to be added to the IDE, this factor is not limited only to the version of the IDE, but also includes extensions. Table 2 provides some sample values of plugins that may be added to the base Eclipse IDE installation.

In an IDE, different user interface elements with which a developer may interact are visualized on the screen. Most often, these are different views and editors that allow developers to work on different project artifacts or provide additional related information, such as workspace or project structure, console output, or list of warnings. While the set of installed plugins is a relatively stable contextual factor, user interface elements represent the contemporary state of an IDE, thus we included active user interface elements contextual factor in our model as well.

Through the user interface elements of an IDE, a developer interacts with particular project artifacts. The user interface elements used to access information determine usable functionality. For example, in Eclipse, the Project Explorer user interface element displays the hierarchical structure of files and enables a developer to create, move, rename, and delete them; in an editor, a developer can edit the content of the files. We use the user interface element with focus contextual factor to capture where interactions are occurring in the IDE.

4. Model application

To be used, a context model has to be adapted to a specific IDE and the data set structured according to the model has to be properly populated. Adapting the model involves determining the set of possible values, for each contextual factor. In the previous section, we provide some examples of particular contextual factor domains for one particular IDE, the Eclipse IDE (e.g., Table 2). However, each IDE, such as VisualStudio or IntelliJ IDEA, will have a specific adaptation of the context model given different characteristics of the IDE and data collection tools used to populate the context model data set.

We can define some contextual factors in our model uniformly across all IDEs. Four contextual factors of this type are developer unique identifier (who), developer general experience (who), time of the day (when), and day of the week (when). For other contextual factors, we had to either 1) split a factor into sub-factors to manage the domain size, 2) adapt the domain, or 3) define IDE specific thresholds. We show the examples of these adaptations in the second part of the section; before describing how we adapt the remaining nine contextual factors for Eclipse and how we populate the data set, we provide background on characteristics of the Eclipse IDE and data collection tools that we used in our study.

4.1. Background information

We begin with a description of user's perspective of Eclipse. The understanding of the basic concepts of Eclipse is needed to understand how we adapt some contextual factors from the where group (see Section 3.4). We then describe how we collect the data to populate the model at a particular point in time, using three Eclipse plugins.

4.1.1. The Eclipse IDE

The available views, editors, commands, and similar elements in an Eclipse installation depend on the set of installed plugins. Every plugin defines a unique identifier for each user interface element it provides.

User interface elements in Eclipse have the following hierarchical structure: the main workbench window contains one or more perspectives, and at every moment there is one active perspective per window; perspectives contain views and editors, and there can be several views and editors opened at the same time. Developers can switch between different perspectives, they can open and close views, and they can open and close editors. They can also change the location and size of views and editors on the screen. Only one view or editor can be active—have focus—at any moment; not necessarily all opened views and editors are (fully) visible at any moment, since they are usually organized as tabs within a limited part of the screen. Fig. 1 shows an annotated example of Eclipse user interface, indicating its major parts, such as the toolbar, a particular view, and an editor. Fig. 2 shows multiple different views.

4.1.2. Data collection tools

To collect the raw data from which we can extract higher-level context values, we use three monitoring tools that do not require any additional manual effort or changes in the development process after the installation. These are Eclipse UDC, a custom version of Mylyn, which we call Lema, and a new plugin, which we call Context Detector (CD).

---

9 According to our experience, IDE users do not install new or uninstall old plugins regularly, even if they stop using them.

10 http://eclipse.org/epp/usagedata
11 http://www.eclipse.org/mylyn
12 We modified the source code of Mylyn in a way that the data collection is running all the time, which means that task definitions and activations are not required anymore; and it also logs Mylyn events together with the additional data related to user interface elements.
Eclipse UDC collects IDE events, namely, command executions, user interface elements activations, and start and stop events of bundles; Table A.10 in the appendix shows an example of a UDC generated event log. Lema records the artifact with the focus, when the focus is in an editor, active perspective, editors, and views, on every editing or selection event; Table A.11 in the appendix shows an example of a portion of a Lema event log. CD has two functions: it tracks the changes in the resource files included in the developer's workspace and it provides export functionality. The first functionality enables offline calculations of source code metrics without considerably slowing down developer's IDE: CD creates a local, hidden GIT repository to which it commits the content of the workspace, whenever it detects a change in any file included in the workspace, i.e., on every “save file” event, and on “start Eclipse” event. The second functionality was implemented due to the participants’ request to be able to export the data collected during the case study with one click; thus, CD can export the before-mentioned GIT repository, all the data collected by Eclipse UDC and Lema, and Eclipse installation history, which is recorded by Eclipse itself.

From Eclipse UDC logs, we can extract the information about when developers executed IDE commands and how many commands they know, i.e., developer IDE experience. From Lema logs we can extract the information about active user-interface elements,
element with focus, and project artifact with focus, which we use
to extract the type of the artifact under development and calculate
the related source code metrics from the data stored in GIT repos-
itory, to obtain the values of length and complexity of the artifact
under development. By merging Eclipse UDC and Lema logs, we can
identify developer’s current and previous activity. Eclipse installa-
tion history enables the extraction of IDE instance values.

4.2. Adaptation of the contextual factors

A general context model often needs to be adapted for a partic-
ular situation. We explain three types of modifications that may be
needed: modifications of the context model, modifications of the
contextual factor domains, and determination of thresholds for
contextual factor domains.

4.2.1. Modifications of the context model

If the contextual factors presented in a general model are very
abstract, it may be hard for a context-aware tool to make ef-
eective use of the factor. In these cases, the contextual factor
may need to be split into more specific sub-factors. For exam-
ple, it is hard to model the active user interface elements (where)
in Eclipse as Eclipse’s user interface elements have a hierarchical
structure. For this contextual factor in Eclipse, we map the factor
to two more specific contextual factors: active perspective con-
textual factor, which represents the top-level user interface element
in Eclipse, and opened views and editors contextual factor, which
represents a set of lower-level user interface elements, in the two-
level hierarchical structure.

4.2.2. Definitions and modifications of the contextual factor domains

As already mentioned, certain domains of contextual factors
presented in our model directly depend on the chosen IDE, and
thus have to be explicitly defined before the model is applied in
practice.

To represent our context model for Eclipse, we consider that
each unique set of installed Eclipse plugins represents one possible
value of IDE instance (where) and each unique set of opened views
and editors represents one possible value of opened views and ed-
itors (where). Domains of active perspective (where) and user inter-
face element with focus (where) contextual factors consist of unique
identifiers of these elements.

The domain of type of the artifact under development (what)
consists of the suffixes of opened files, in addition to “Java class”
and “Java method”. We chose this approach over the file level, as
did Singer et al. (2005), because artifacts available within the IDE
often have special internal structure, as when object-oriented code
is shown via its class, method, and field constituent parts. Future
work might consider broader and more detailed domain for type of
the artifact under development factor; at the moment we use this
domain, because our data collection tools are only able to detect
these artifacts.

To be able to automatically identify development activities, we
also modified the domains of current activity (what) and previous
activity (what) contextual factors. Since we are not able to dis-
tinguish between reading activity, reviewing activity, breaks, and
“other”, without physically observing developers, we categorize all
time intervals of sustained inactivity in the IDE that are longer
than three minutes as breaks\(^\text{13}\) and time intervals that are shorter
than three minutes as part of the activity. With our monitoring
tools, we can detect:

- editing,
- navigating,
- debugging,
- use of version control, and
- other.

4.2.3. Definitions of the contextual factor domains’ thresholds

Similarly to contextual factor domains, certain thresholds can
depend on, or be affected by, a specific IDE. For example, to classify
developers according to developer IDE experience (who), we use the
number of different Eclipse commands used by a developer. Since
the average Eclipse user is using 42 different commands (Murphy-
Hill, 2012),\(^\text{14}\) we classified those that are using more than 42 as
experts, and those that are using 42 or less as novices.

To distinguish between short, medium, and long and sim-
ple, medium, and complex artifacts under development, we use
lines of code (LOC), cyclomatic complexity (CC) (McCabe, 1976),
and weighted methods per class (WMC) (Chidamber and Ke-
merer, 1994) metrics. Our mapping of computed metric values to
the contextual factor domain is presented in Table 3. The thresh-
olds are similar to those used by Mäntylä and Lassenius (2006).

These metrics are sufficient as we currently extract only the val-
es of length of the artifact under development (what) and complex-
ity of the artifact under development (what) when type of the artifact
under development (what) is either “Java class” or “Java method”,
due to the limitations of the monitoring tools.

4.3. Population of the data set

Each contextual factor must be computed at each moment in
time to populate the context model data set so that the informa-
tion can be used by context-aware tools and applications. Table 4
describes the approaches we use to populate each contextual factor
included in our model. The implementation of the data set is not
limited to a specific technology or format; for the purposes of our
analyses, we stored the values of contextual factors in a relational
database and “comma-separated values” (CSV) files.

5. Evaluation of the context model

To evaluate the context model factors, we chose a common sce-
nario of IDE use in which the users are interacting with an IDE by
executing its commands. We chose to evaluate the model based on
the analyses of statistical correlations between the factors and
command executions. If the command executions change signifi-
cantly when the situation—as detected by the contextual factor—
changes, it is highly unlikely that different command executions in
different situations are purely coincidental. Which means that
such a type of knowledge about the situation is informative and it
can be exploited in applications that use context to improve the
interaction between users and an IDE. To perform the evaluation,

\(^{13}\) Three minutes timeout is used also in Mylyn, according to Kersten and Mur-

\(^{14}\) Murphy-Hill (2012) analyzed publicly accessible data collected from hundreds
of thousands Eclipse users to calculate the average number of different commands
used by an average user.
we conducted a case study and used the statistical framework suggested by Odic et al. (2013).

The results show that all the factors included in the model contribute to more accurate identification of situations in which developers are more inclined or less inclined to use certain commands. In this section, we present the details of the conducted case study and statistical tests, together with the results and their validity threats.

5.1. Participants

Eight participants were included in the case study. All of them primarily develop in Java; however, as we could see from the data, they are using Eclipse also for other purposes, such as writing documents in Latex.

Four participants are software developers in a medium-sized company, the other four work in academia as researchers or research assistants. One participant is a woman. Their age ranges from 25 to 40, and they have between 1 and 10 years of experience using Eclipse.

5.2. Data

We asked eight participants to install Eclipse UDC, Lema (custom version of Mylyn plugin), and CD (our own plugin), and collected data for ten weeks. After we received the exported data sets from all participants, we filtered the recorded events and excluded those that are due to the use of monitoring tools, such as data export events and Mylyn events. Then we extracted, for each logged command execution, the values of every contextual factor.

Participants executed more than 100,000 commands during 10 weeks. When we observe the distribution of command executions detected during the case study presented in Fig. 3, we can see that it has a long tail: 193 different commands were executed across all participants, 15 most popular commands represent 80% of executions and 65 represent 99% of executions. Several commands are only executed in very special situations. The most extreme case is the Synchronize All command, which was executed only when the active perspective was Team Synchronizing and was the only command executed in this context. Tables 5 and 6 provide more specific information about the properties of this data set.16

5.3. Statistical tests

We performed statistical tests to see whether the contextual factors included in our model correlate with the usage of commands. We used Pearson’s $\chi^2$ test and Fisher–Freeman–Halton test, if the power of the $\chi^2$ test was too low or Cochran’s rule was not satisfied, to evaluate the significance of the differences in the command executions when the values of the factors change. Odic et al. (2013) use the same statistical tests, observing the

\[ \chi^2 \]

\[ \text{df} \]

\[ p \]

\[ \text{Expected count} \]

\[ \text{Observed count} \]

\[ \text{Cochran’s rule} \]

\[ \text{Significance} \]

\[ \text{Independence} \]

\[ \text{Association} \]

\[ \text{Contingency} \]

\[ \text{Table} \]

\[ \text{Figure} \]

\[ \text{Equation} \]

\[ \text{Property} \]

\[ \text{Method} \]

\[ \text{Result} \]

\[ \text{Conclusion} \]

\[ \text{Future work} \]

\[ \text{Discussion} \]

\[ \text{Limitations} \]

\[ \text{Conclusion} \]

\[ \text{Acknowledgments} \]

\[ \text{References} \]

\[ \text{Appendix} \]

16 The list of command executions with the corresponding contextual factor values is publicly available at: http://sites.google.com/site/mgasparic/context.

17 Cochran’s rule is satisfied when in a contingency table at least 80% of the cells have the expected count higher or equal to five and all the expected counts are higher or equal to one.
changes of explicit movie ratings, to evaluate the context model later implemented in a movie RS. They show that certain contextual factors correlate with the ratings, while others do not. We use statistical tests to identify which contextual factors correlate with the executions of which command.

Hence, the only difference between our tests and those used by Odic et al. is that they use item rating when we use item consumption, which is in our case command execution. Ratings express user evaluations of the items. If a contextual factor is correlated with item ratings, then this signifies that the user’s rating behavior, i.e., her preferences, changes depending on the context. In our case, if a contextual factor correlates with the usage of a command this means that the user is using the command differently, when the values of the contextual factor change.

From 14 contextual factors and 193 commands, we generated 2702 contingency tables, in which the two variables were contextual factor value and command executions, which is a binary variable with value “true” when the observed command is executed and value “false” when any other command is executed. Cells of contingency tables contain the number of command executions. A contingency table example is shown in Table 7.

We generated contingency tables and conducted statistical tests for the entire sample to be able to observe overall inclination of developers to execute certain commands in different situations. Additionally, we performed the same analysis for each participant, to be able to observe the inclinations to the command executions on the individual level and to exclude the risk of observing only the differences between developers instead of observing the differences between the contextual factor values.

The null hypothesis used in all our tests is: contextual factor and the usage of a command are independent. The alternative hypothesis is: they are dependent. To calculate p values we used R software environment and built-in functions (chisq.test and fisher.test). In Fisher–Freeman–Halton test we used Monte-Carlo simulation with the sample size 10,000. Low p values indicate that the probability that the contextual factor values and command usages are independent is low. When p value is less than 0.05, the null hypothesis is likely wrong. To be able to assess the accuracy of our tests, we performed post-hoc power analyses. In Pearson’s $\chi^2$ test we used the effect size 0.2 and significance level 0.05 and in Fisher–Freeman–Halton test we used Monte-Carlo method.

### 5.4. Results

Statistical analyses show that for the entire sample, as well as for each of the eight individual participants, when he or she worked in different situations and executed enough commands to allow reliable conclusions (i.e., the power of the test is above 0.95), the usage of certain commands and contextual factors are dependent variables; however, we note that the set of the commands that correlate with different values of the factors varies between the factors and between the participants. This is true for all fourteen factors included in our model.

Table 8 displays the results of the tests. The entries shown with a checkmark (✓) represent the number of commands for which the power of the statistical test is above 0.95 and p value is below 0.05. In these cases, the null hypothesis can be rejected and we can conclude that the change in command usage is significant when the value of the factor changes. The entries shown with a question mark (?) represent the number of commands for which the power of the test is sufficiently high but p value is above 0.05 and the entries shown with a question mark (?) represent the number of commands for which the power of the test was below 0.95. In these two cases, we cannot reject the null hypothesis.

The evaluation of developer unique identifier, developer general experience, and developer IDE experience in relation to each individual developer is meaningless since the contextual factors depend on the developer and did not change during the study execution. For five participants (A, B, C, D, and E), we could not compare different values of the IDE instance contextual factor, as these participants did not install new plugins, thus their IDE instance did not change during the study. For one participant (E), we could not compare the values of active perspective, as the participant used only one perspective during the entire study. And, we were not able to obtain the data needed to extract the values of length of the artifact under development and complexity of the artifact under development contextual factors for participants A, B, and H, because GIT repositories became corrupted during the study. All these entries are shown with a dash (−) in Table 8.

By observing the entire sample, we can see that the contextual factor that is correlated with the largest number of commands is developer unique identifier; this indicates that the systems supporting users in their interaction with an IDE should be personalized, because users behave differently, even if we observe only a small set of users working in a similar environment. Almost the same result is achieved by IDE instance; probably because only three participants changed it during the case study and consequently it strongly reflects developer unique identifier, on the level of the sample. On the third place, we find current activity and opened views and editors. The first is expected, since commands are strictly connected to the current activity: a developer who executed Delete Line command is currently editing and the one who executed Re-

<table>
<thead>
<tr>
<th>Table 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic properties of the data set used in the case study.</td>
</tr>
<tr>
<td>Number of participants</td>
</tr>
<tr>
<td>Study duration</td>
</tr>
<tr>
<td>Number of all command executions</td>
</tr>
<tr>
<td>Number of command executions per developer in 10 weeks:</td>
</tr>
<tr>
<td>A</td>
</tr>
<tr>
<td>7,069</td>
</tr>
<tr>
<td>Average number of command executions per developer per day:</td>
</tr>
<tr>
<td>A</td>
</tr>
<tr>
<td>101</td>
</tr>
<tr>
<td>Total number of different commands</td>
</tr>
<tr>
<td>The most used command</td>
</tr>
<tr>
<td>(13,628 executions, 43%)</td>
</tr>
<tr>
<td>The least used command</td>
</tr>
<tr>
<td>Number of commands used by all</td>
</tr>
<tr>
<td>Number of different commands used per developer:</td>
</tr>
<tr>
<td>A</td>
</tr>
<tr>
<td>60</td>
</tr>
</tbody>
</table>
Table 6
Three most common values of contextual factors and corresponding numbers of command executions.

<table>
<thead>
<tr>
<th>Contextual factor</th>
<th>Domain size</th>
<th>Values</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Developer general experience</td>
<td>3</td>
<td>competent</td>
<td>72,728 exec. 69.4%</td>
</tr>
<tr>
<td>Developer IDE experience</td>
<td>2</td>
<td>expert</td>
<td>104,036 exec. 99.9%</td>
</tr>
<tr>
<td>Current activity</td>
<td>5</td>
<td>edit</td>
<td>56,402 exec. 53.8%</td>
</tr>
<tr>
<td>Previous activity</td>
<td>6</td>
<td>edit</td>
<td>67,728 exec. 64.6%</td>
</tr>
<tr>
<td>Type of the artifact</td>
<td>12</td>
<td>Java method</td>
<td>68,274 exec. 65.2%</td>
</tr>
<tr>
<td>Length of the artifact</td>
<td>4</td>
<td>undetected</td>
<td>52,529 exec. 50.1%</td>
</tr>
<tr>
<td>Complexity of the artifact</td>
<td>4</td>
<td>undetected</td>
<td>52,529 exec. 50.1%</td>
</tr>
<tr>
<td>Time of the day</td>
<td>4</td>
<td>afternoon</td>
<td>64,275 exec. 61.3%</td>
</tr>
<tr>
<td>Day of the week</td>
<td>7</td>
<td>Tuesday</td>
<td>30,052 exec. 28.7%</td>
</tr>
<tr>
<td>IDE instance</td>
<td>13</td>
<td>IDE used by E</td>
<td>32,015 exec. 30.6%</td>
</tr>
<tr>
<td>Active perspective</td>
<td>7</td>
<td>Debug</td>
<td>35,849 exec. 34.2%</td>
</tr>
<tr>
<td>Opened views and editors</td>
<td>372</td>
<td>Set of 14 UI elements18</td>
<td>13,442 exec. 12.8%</td>
</tr>
<tr>
<td>UI element with focus</td>
<td>44</td>
<td>Compilation Unit editor</td>
<td>71,090 exec. 67.8%</td>
</tr>
</tbody>
</table>

Table 7
Contingency table of developer A for the type of the artifact under development contextual factor and Paste command.

<table>
<thead>
<tr>
<th>Type of the artifact under development</th>
<th>Number of Paste executions (i.e., true)</th>
<th>Number of other executions (i.e., false)</th>
</tr>
</thead>
<tbody>
<tr>
<td>undetected</td>
<td>0</td>
<td>185</td>
</tr>
<tr>
<td>.java</td>
<td>33</td>
<td>311</td>
</tr>
<tr>
<td>.map</td>
<td>7</td>
<td>19</td>
</tr>
<tr>
<td>.properties</td>
<td>16</td>
<td>184</td>
</tr>
<tr>
<td>Java class</td>
<td>58</td>
<td>361</td>
</tr>
<tr>
<td>Java method</td>
<td>797</td>
<td>5098</td>
</tr>
</tbody>
</table>

so large: it contains 372 different values, which is more than all other factors combined. Consequently, we conclude that the correlations are more likely if the factor has a larger domain, however, the generalizability of the situations may suffer if every situation is different, as we already discuss in Section 3.

The only factor that performs very poorly on our data set is developer IDE experience. This is probably due to the size of the set of participants and their characteristics, since we only classified participant H as a “novice” and he only executed 15 different commands. For 10 of those commands, the null hypothesis can be rejected, but for others it cannot be, mostly because 178 commands were never executed by a “novice”. Eventually, this may also be related to the fact that the experience determines the set of the commands that are being used by a particular user, but it does not strongly influence the number of the executions of the known commands, which is measured by our analysis. However, to reliably examine the correctness of such conclusions, additional tests with a greater sample, particularly including more novices, would have to be performed.

The set of executable commands and the visualization of menu bars in Eclipse depend mainly on the user interface element with focus and type of the artifact under development. Many of the factors

---

18 Compilation Unit editor and Breakpoint, Debug, Variable, Palette, Result, Package Explorer, Search, Console, Content Outline, Problem, Progress, Task List, and Servers views.
19 Compilation Unit editor and Breakpoint, Debug, Variable, Result, Package Explorer, Search, Console, Content Outline, Problem, Task List, and Servers views.
20 Compilation Unit editor and Bug Info, Bug Tree, Debug, Search, Console, Project Explorer, All Markers, Content Outline, Progress, and Servers views.
Table 8  
The results of statistical tests showing contextual factors and corresponding number of commands for which p value is below 0.05 and power is above 0.95 (√), p value is above 0.05 (×), and power is below 0.95 (?).

<table>
<thead>
<tr>
<th>Contextual factor</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>All</th>
</tr>
</thead>
<tbody>
<tr>
<td>Developer unique identifier</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>137</td>
</tr>
<tr>
<td>Developer general experience</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>118</td>
</tr>
<tr>
<td>Developer IDE experience</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>10</td>
</tr>
<tr>
<td>Current activity</td>
<td>√ 41</td>
<td>√ 77</td>
<td>√ 62</td>
<td>√ 39</td>
<td>√ 53</td>
<td>√ 47</td>
<td>√ 37</td>
<td>√ 4</td>
<td>130</td>
</tr>
<tr>
<td>Length</td>
<td>× 19</td>
<td>× 16</td>
<td>× 23</td>
<td>× 27</td>
<td>× 28</td>
<td>× 23</td>
<td>× 23</td>
<td>× 0</td>
<td>63</td>
</tr>
<tr>
<td>Previous activity</td>
<td>× 7</td>
<td>× 0</td>
<td>× 7</td>
<td>× 0</td>
<td>× 7</td>
<td>× 0</td>
<td>× 7</td>
<td>× 0</td>
<td>7</td>
</tr>
<tr>
<td>Type of the artifact</td>
<td>√ 30</td>
<td>√ 48</td>
<td>√ 40</td>
<td>√ 36</td>
<td>√ 45</td>
<td>√ 40</td>
<td>√ 19</td>
<td>√ 2</td>
<td>101</td>
</tr>
<tr>
<td>Complexity of the artifact</td>
<td>× 30</td>
<td>× 45</td>
<td>× 45</td>
<td>× 30</td>
<td>× 36</td>
<td>× 30</td>
<td>× 41</td>
<td>× 13</td>
<td>92</td>
</tr>
<tr>
<td>Time of the day</td>
<td>× 7</td>
<td>× 0</td>
<td>× 7</td>
<td>× 0</td>
<td>× 7</td>
<td>× 0</td>
<td>× 7</td>
<td>× 0</td>
<td>7</td>
</tr>
<tr>
<td>Day of the week</td>
<td>√ 20</td>
<td>√ 32</td>
<td>√ 11</td>
<td>√ 22</td>
<td>√ 14</td>
<td>√ 12</td>
<td>√ 10</td>
<td>√ 0</td>
<td>72</td>
</tr>
<tr>
<td>IDE instance</td>
<td>× 40</td>
<td>× 61</td>
<td>× 20</td>
<td>× 44</td>
<td>× 29</td>
<td>× 16</td>
<td>× 50</td>
<td>× 0</td>
<td>121</td>
</tr>
<tr>
<td>Active perspective</td>
<td>× 7</td>
<td>× 0</td>
<td>× 7</td>
<td>× 0</td>
<td>× 7</td>
<td>× 0</td>
<td>× 7</td>
<td>× 0</td>
<td>7</td>
</tr>
<tr>
<td>Opened views and editors</td>
<td>√ 24</td>
<td>√ 41</td>
<td>√ 10</td>
<td>√ 19</td>
<td>√ 43</td>
<td>√ 36</td>
<td>√ 18</td>
<td>√ 4</td>
<td>130</td>
</tr>
<tr>
<td>UI element with focus</td>
<td>× 3</td>
<td>× 52</td>
<td>× 75</td>
<td>× 47</td>
<td>× 1</td>
<td>× 0</td>
<td>× 10</td>
<td>× 71</td>
<td>0</td>
</tr>
</tbody>
</table>

5.5. Validity threats  
A number of threats affect the results of our case study.  

5.5.1. Conclusion validity  
Conclusion validity is related to the influences that can affect the correctness of conclusions about relations between the treatment and the outcome (Wohlin et al., 2012). In our case study, construct validity is related to the assumptions about the relevance of contextual factors.

We minimized this type of threat by applying appropriate statistical tests and by acquiring a large enough data set, as can be seen from the power values of the post-hoc power analyses presented in the previous section. Furthermore, we lowered the risks related to random heterogeneity of participants by selecting relatively similar developers, who are all primarily using Eclipse to program in Java.

To obtain reliable measurements, we mainly used well tested software to collect the data. Additionally, we manually inspected the collected data, and we excluded parts of data sets that included in our model correlate with more commands than those two factors. Hence, it seems that the adaptation of the menu bars and executable commands could be better contextualized if additional context was taken into account.

By observing the results of the tests on the level of individuals, we can also see that all case study participants tend to change the usage of commands when their situation changes. In Table 9, we show some examples of different situations in which developers become more inclined or less inclined towards the execution of different commands. We present the percentage of average executions of five most used commands and five arbitrarily selected commands when the most common values of the contextual factors were detected. For instance, the Organize Imports command was executed 1810 times in the set of all 104,789 command executions, which is approximately 1.73%, but on Tuesday, the proportion of Organize Imports command executions is higher by almost 1%, which means that the participants add new and remove unused imports a lot more often on Tuesday than on average.
contained incomplete or corrupted data. However, there can be failures in context detectors that we were not able to detect during the inspections. Murphy-Hill et al. (2012) noted that Eclipse UDC does not detect all command execution events, and Soh et al. (2015) discovered that average Mylyn log lacks about 6% of the time spent on a task and contains about 28% of false edit-events. We were not able to address this threat, since we were not able to reliably fix existing tools or develop from scratch the new ones, due to the lack of time and expertise. However, we are aware that it is possible that up to 28% of previous activity contextual factor values were wrongly identified as edit-events, which could influence the evaluation of the relevance of previous activity contextual factor, and that it is possible that we missed the executions of certain commands, which could influence the evaluation of the relevance of all contextual factors. Nevertheless, it is unlikely that the detection of additional edit-events and additional command executions could considerably raise the p values. In our data, we observed the opposite trend: more command executions in a contingency table decrease the p value.

5.5.2. Internal validity

Internal validity is related to the influences that can affect the correctness of conclusions about causal relationships between the treatment and the outcome (Wohlin et al., 2012). Similar to the conclusion validity threats, these threats are related to the assumptions about the relevance of contextual factors.

Since our sample of developers is small, the direction of causal influences can be ambiguous. Especially, there is a risk that different developers work in different situations, which would mean that during the analyses of the overall data, we would not observe the differences in IDE command usage that are due to different situations, but we would observe the differences in IDE usage between developers. For instance, the obvious difference is a working environment, since four participants work in a company as software developers, while other four work in academia as researchers; consequently, they could be using Eclipse for a different purpose. We addressed this threat by performing the tests of contextual factors relevance for each individual developer.

5.5.3. Construct validity

Construct validity is related to the influences that can affect the correctness of generalization of the results to the concept or theory behind the study (Wohlin et al., 2012). In our case, it is related to the assumptions about the quality of the theoretical model presented in Section 3, based on the results of the case study.

Since we evaluated the model by analyzing command usage of a small and particular set of software developers, the construct of the study is under-represented. Furthermore, because we used only one measure to assess the relevance of each contextual factor, except for the active user interface elements factor, where we used two measures, there is a risk that the results are misleading due to the measurement bias.

Additionally, our case study is exposed to social construct validity threats, since people, when being observed, tend to change their behavior, to look better (Wohlin et al., 2012). We detected the inclination of participants to provide as much data as possible. It seems that they were trying to use Eclipse more often than they would use it otherwise, even though we made an explicit request to keep the same working habits as they had before and that we conducted long term, fully automated data collection.

We are also aware of the deviations in IDE usage due to preliminary analysis, during which the users had to keep activating Mylyn tasks. As already mentioned, we excluded the explicit events in the logs, however, we are not able to assess the indirect impact of these actions on the overall usage of the IDE.

5.5.4. External validity

External validity is related to the influences that limit the generalization of the results to industrial practice (Wohlin et al., 2012).

To improve the external validity of our case study we included professional developers from medium-sized company. However, our sample is not representative, since these developers mainly use Java programming language, they use only Eclipse IDE, and they are using a significantly bigger set of different commands than an average Eclipse user. Consequently, it is possible that if we applied our model to characterize the situations of different groups of developers, such as novices or VisualStudio users, for example, certain contextual factors would be irrelevant.

6. Discussion and future work

This paper presents a new context model that can be used to describe situations of developer's using an IDE. We discuss issues
associated with our choices for contextual factors included in the model, potential improvements of the existing RSSEs that can be achieved by taking into account more comprehensive context models, application of the context model to a particular IDE, and our approach to assessing relevance of the contextual factors.

6.1. Context model

We created a model that can be used to characterize a developer’s situation while she is using an IDE. This model contains more contextual factors than the existing models used in applications supporting software development, such as RSSEs, and characterizes developer’s situation from different perspectives, by addressing all four basic context questions, i.e., who, what, when, and where. Nevertheless, there are many possibilities for further extensions and modifications.

Future work may consider different factors, scales, and approaches to characterize developer’s experience, such as developer activity experience, for example. Some of the alternative approaches to evaluate the level of experience are described by Feigenspan et al. (2012). New models may consider other classifications of activities, as well as different temporal perspectives, such as previous five activities or activities in the session. They may include tasks, i.e., atomic and well-defined work assignments (Maalej et al., 2014), which we replaced with activities, because the activity detection does not require manual effort, we can assume higher accuracy, and they are more generalizable, as already discussed above. Future work may also consider to include contextual factors that describe artifacts with other meta-information, such as those described by Gómez et al. (2008) and Kitchenham (2010).

Future work may consider different temporal scales, such as working days versus non-working days, as well as different partitions of the day. Moreover, future models may consider not only temporal aspects in when category, but also process-oriented aspects, such as phase of a project. It could be beneficial to use semantic annotations for user-interface elements and IDE plugins, instead of unique identifiers, as we do in this paper. For instance, we could group these elements as we group interaction events into “editing”, “debugging”, “using version control”, etc. In that way, the generalizability of active user interface elements, user interface element with focus, and IDE instance contextual factors could be improved, and the domain size could be decreased as well.

New models may try to characterize how developers interact with other tools outside an IDE and how developers work with colleagues in an organization, amongst others. They may also examine the usefulness of more detailed information in a more specific application, compared to our relatively general model.

We think that it would be beneficial if new, competitive, and fundamentally different context models would emerge in the future, so that the notion of the context in software engineering could be studied from different perspectives. For instance, previous research in other disciplines showed that humans, and also software applications, can understand the context in several ways. According to Dourish (2004), conventional interpretation of context assumes that the definition of the context is a representational problem, i.e., we can assume that context is stable, in a sense that its representation may vary from application to application, but not from instance to instance of an activity or an event, and we can assume that it is possible to redefine what counts as the context of activities, since activities happen within the context. Our model is an example of research based on these assumptions: a certain developer interacts with an IDE when she is performing some working activities, in a certain time interval, and in a certain type of environment.

Nevertheless, the definition of context can be understood also as an interactional problem. Instead of assuming that the context is around the activity, we can assume that it arises from the activity, thus contextual factors have to be defined dynamically for each particular activity or event (Dourish, 2004). Certain context models used in the systems mentioned in Section 2 comply with the interactional point of view. For example, the Spyglass tool observes the behavior of the user and analyzes interaction sequences to find out whether certain steps could be replaced by IDE commands (Viriyakakittiaporn and Murphy, 2010).

Adomavicius et al. (2011) provide another categorization for context models. They defined six types of context that are different according to two dimensions: how contextual factors change over time and what is the knowledge of the system about contextual factors. With respect to change over time, they distinguish between static and dynamic context models: the model is static if the set of relevant contextual factors and their domains remain the same over time and the model is dynamic if they can change. According to this definition, we can classify our model as dynamic, since domains of certain factors, those that are marked with “a” in Table 2, can change over time.

The knowledge of the system about contextual factors can be split in three categories:

- fully observable: all relevant contextual factors are known explicitly;
- unobservable: the behavior of the observed entity depends on the context, but the system does not know which contextual factors affect it; and
- partially observable: the system has some explicit information about the contextual factors.

If we knew that all the contextual factors presented in our model affect the use of IDE and that these are the only relevant factors, our model would be fully observable. If we did not know a single contextual factor that affects the behavior of the IDE user, for our systems the contextual factors would be unobservable. Since neither is true, i.e., we know for sure that the executions of at least some commands correlate with all the factors included in our model, we can classify our contextual factors as partially observable. Even though it is highly unlikely that we can acquire the complete context (Dey, 2010), the future work should aim to improve the observability of the factors.

6.2. Application of the model in practice

We envisage our model to be used in context-aware systems for the purposes of user support in IDE usage. In this paper, we show how we can automatically collect certain data in Eclipse. We believe that the designers of the context-aware systems that are targeting more general populations of developers will have to overcome more difficulties than we had to during the case study we performed. For example, the obvious obstacle to extract the values of certain contextual factors is related to privacy of the developers and confidentiality of their working processes, as well as products (e.g., source code). We did not want our case study participants to be intimidated by any kind of privacy concerns.21 and due to that, we did not record the screen, for example, which would enable more detailed analysis of the activities. On the other hand, we were able to obtain the entire snapshots of workspaces, which were saved in GIT repositories by our CD plugin, including proprietary source code. We are not aware of any mechanism that would

---

21 In one of our previous industrial case studies, which is partially documented by Gasparic et al. (2015), we recorded every focus change on the level of operating system, and we noticed that developers uninstalled our software or started using other computers.
allow the calculations of source code metrics on an average workstation without considerably slowing it down; and we know that in such situations the developers would tend to stop using the tool (Gasparic et al., 2013). Thus, the practical use of some contextual factors presented in our model is limited to retrospective analysis, which are not always possible, since, in our opinion, most of the Eclipse users will not share the data with the authors of IDE plugins.

Another important aspect concerning the implementation is the amount of manual effort required to use a system. For example, we manually encoded contextual factor values of developer unique identifier and developer general experience. We were able to do that because all case study participants used only one workstation, one Eclipse installation, and one workspace, and we were able to communicate with them. If that was not the case, we would need to implement login and registration functionality, that would enable the collection of the data required to extract the values of developer unique identifier and developer general experience contextual factors, but would also require additional effort from the developers. In addition to these, there are many other potentially relevant contextual factors that may require additional manual effort from the developer. For instance, Mylyn users have to define and activate tasks that they are working on, in order to provide the context needed by the system (Kersten and Murphy, 2006). In our opinion, in that way the context is not only monitored, but also changed. Consequently, there might exist a completely different category of relevant contextual factors which are related to the user’s interaction with a context-aware application, especially when this application is proactive or requires a lot of manual effort (Gasparic and Ricci, 2015).

To the best of our knowledge, the model presented in this paper is the first context model that has been designed particularly for characterizing general situations of a developer using an IDE. In the following sections, we discuss the potential of contextual factors included in our model to improve the accuracy and timing of recommendations produced by the RSSEs described in Section 2, which largely rely on single-factor context models or describe the situation only from one perspective.

6.2.2. Adapting the user interface of an IDE

Once the accuracy of the tools like Mylyn (Kersten and Murphy, 2006) and AutumnLeaves (Roethlisberger et al., 2009) is high enough, automatic adaptation of all user interface elements could become a core feature of the IDEs; as Gu et al. (2014) envisaged in their next generation. Moreover, it can even be extended to the entire operating system, as Kersten (2007) and Maalej and Sahm (2010) suggest. However, the usability of Mylyn is hampered because half of the software developer’s work is not defined as tasks (Maalej, 2009). We suggest to replace the tasks with activities and try to improve the accuracy of detecting relevant artifacts by taking into account additional information, such as which views are opened, what are the current artifacts under development, and time period. Minelli et al. (2014) discovered that certain developers use IDE user interface elements and project artifacts in a sequential manner, and that windows supporting debugging and inspection are short-lived, compared to the windows used during the development. Such information could be used also in AutumnLeaves, to identify the users that are unlikely going to revisit an opened window or assign lower weights to windows opened during debugging, for example.

6.2.3. Adapting the order of recommendations

As has been shown by OCompletion (Robbes and Lanza, 2010) and Quick Fix Scout (Mušl et al., 2012), taking into account additional information about the activities can improve the order of recommendations already produced by an IDE, as well. Such improvements are important, since navigating over long lists of recommendations or undoing wrong automated transformations, can be more time consuming than manually implementing the suggestions. We believe that OCompletion and Quick Fix Scout could be improved even further if they would be more personalized. For instance, it may be hard for novices to understand certain suggestions, such as unknown API calls or complex refactorings, and they will have to spend a lot of time to understand them and compare them with others. Thus, if the objective is to improve developers’ performance, it could be beneficial to take into account also their knowledge.

6.2.4. Providing relevant examples

The importance of more comprehensive context models has been expressed also in the fields that are not directly related to the general interaction with an IDE, such as source code reuse (Holmes et al., 2006; Hummel et al., 2007; Janjic et al., 2014). The motivation for more rich queries than those used by Holmes and Murphy (2005), is that simple queries often lead to poor results (Hummel et al., 2007). We believe that distinguishing between different artifacts, based on the type of the interaction, could improve the quality of recommendations even further; for instance, source code that is being edited probably represents developer’s task better than the source code through which the developer only navigated, as also the results of Minelli et al. (2014) indicate. Furthermore, we can imagine scenarios of IDE interactions which could provide additional clues about the meaning of the developer’s actions and could lead to more relevant recommendations. For instance, certain developers use debugging tools, such as Debug perspective, Variable view, or just Console view, when they are having problems; we assume that in such a case, code reuse tools should show the snippets of proper API usage that are similar to the code under development. On the other hand, if

22 A sceptical reader could say that this would be inconvenient, since the IDE user would get lost everytime she would try to fire a command. Thus, an elegant so-
the system would detect that the developer is reimplementing a functionality contained in the API, which is a bad practice, it would be more useful if the system would show the snippets that can replace the written source code.

6.2.5. Improving the timing of recommendations

Finally, after RSSEs generate useful and well-ordered recommendations, they have to deliver them at the right moment. It can happen that the harmfulness of the interruption outweighs the usefulness of the recommendation, but certain recommendations are valuable for the user only if they are presented at the right time (Murphy-Hill and Murphy, 2014). For instance, Spy-Glass (Viriyakattiyaporn and Murphy, 2010) suggests tools for navigation: if the developer finishes the task manually, before the recommendations are delivered, they can only become useful in the next session.

It seems that broader context could provide more accurate detection of interruptibility; for example, not all project artifacts demand the same amount of cognitive effort: we would imagine that a developer is more interruptable when she is implementing an entity class than when she is debugging a complex method. Furthermore, it could be that the developer is always in a hurry on certain time, such as Monday mornings or Friday afternoons, and is therefore less interruptable at that time. Or it may be that the developer simply does not like to be interrupted after a break, when she is trying to regain the previous mental context. In the case study reported by Carter and Dewan (2010), one participant said that status changes hurt his ego, since slower progress did not occur due to his lack of skills, but due to the complexity of the problem; another participant complained about frequent false positives while building a new product. It seems that the timing of status changes could be improved if the system would take into account the type and the complexity of the artifacts under development, as well as developer’s knowledge.

6.3. Evaluation

The evaluation presented in this paper is focused on the assessment of the potential of the suggested contextual factors to provide meaningful information when characterizing different situations. The results show that the factors do provide such information, since they correlate with the usage of different commands. However, evaluations of this type cannot be used to estimate the usefulness of a model when it is integrated in a certain context-aware application. After the initial evaluation of a context model is performed, designers of applications built on top of the model should test whether the model improves the performance of the system, according to their own metrics, e.g., are recommendations more accurate if context is taken into account?, is the interaction with the system improved?, do developers perceive the outputs more useful?, etc. Furthermore, the designers have to assess the costs of the data collection needed to obtain contextual factor values and gains of the additional information. In the future, we plan to develop a context-aware IDE command recommender system, based on our model, which we will use to evaluate the improvement in the acceptance of recommendations by developers, compared to the existing command recommender systems, e.g., (Murphy-Hill et al., 2012) and (Zolaktaf and Murphy, 2015).

Despite the limitations of our model, it provides broader information than any of the models previously developed in the software engineering community. The evaluation showed that context affects the usage of IDE commands on the individual level, as well as on the group level. It is important to perform an evaluation of a context model, since irrelevant factors can result in unnecessary data collection, which is usually expensive, or can even decrease the performance of a context-aware system, as is the case in the movie recommender system developed by Odic et al. (2012), where irrelevant contextual factors degrade the accuracy of rating predictions. In the future, we plan to evaluate the model also with a bigger and more diverse group of IDE users, to improve the external validity of our conclusions, however, it is hard to believe that the context will have a lower effect on the IDE usage when the diversity of the users and situations will be even greater.

We also plan to assess the importance of each contextual factor using mutual information analysis, since it is possible that some factors presented in our model, in certain situations, provide the same information. For example, if a developer only uses the “De-bug” perspective for debugging, this information is redundant, and since it is computationally cheaper to detect which perspective is opened than to detect the activity, we could lower data collection costs and costs of analysis by omitting activity related factors. A similar case occurred also in our case study, where five out of eight software developers used only one IDE instance and all instances were different, which means that the information provided by IDE instance and developer unique identifier factors is redundant for these five participants. Future work should consider how to adapt the model at runtime, based on, for example, statistical analysis, such as mutual information analysis. In this case, our context model will become interactional, as described in Section 6.1.

7. Conclusions

How developers interact with their IDEs is affected by the situations they are in. In this paper, we present a context model in which a range of context dimensions is addressed, namely, who, what, when, and where. It contains thirteen factors that can be used to characterize and distinguish between different situations in which developers interact with an IDE. The model is not tailored to support the development of a specific application. We show how the model can be adapted to a specific IDE when it is used in practice and we present the results of the evaluation which show that all the factors included in our model statistically correlate with the usage of at least some commands. This type of analysis allows the exclusion of the factors that are not informative based on empirical evidence, which should have been a basic requirement for every context model, before it is introduced in a specific application (Odic et al., 2012; 2013). Yet, this type of analysis does not provide any guarantees that the application built on top of it will benefit from the context model. It is still necessary that the designers of context-aware applications test whether the context model improves the performance of the system, according to their own metrics.

The researchers may want to consider evaluating whether our model improves the accuracy of recommendations generated by many of the proposed RSSEs, such as those developed by Holmes and Murphy (2005); Hummel et al. (2008); Kersten and Murphy (2006); Mușlu et al. (2012), Robbes and Lanza (2010); Roethlisberger et al. (2009), and Viriyakattiyaporn and Murphy (2010), and whether it can help tailor when to provide such recommendations by considering contextual factors to characterize when is a good time for an interruptive recommendation. In the future, we will incorporate our model in a context-aware IDE command recommender system, such as systems presented by Murphy-Hill et al. (2012) and Zolaktaf and Murphy (2015), which aim to increase a user’s use of the breadth of commands in an IDE.

We encourage the community to continue developing new context models that will provide a broad characterization of context for software development to enable appropriate personalized and context-aware support for the wide range of developers and the situations in which they work.
Appendix A. Examples of interaction event logs

Table A.10 Example of Eclipse UDC logs.

<table>
<thead>
<tr>
<th>what</th>
<th>kind</th>
<th>bundled</th>
<th>bundleVersion</th>
<th>description</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>activated</td>
<td>editor</td>
<td>org.eclipse.jdt.ui</td>
<td>3.9.2</td>
<td>value:</td>
<td>org.eclipse.jdt.ui.CompilationUnitEditor</td>
</tr>
<tr>
<td>executed</td>
<td>command</td>
<td>org.eclipse.jdt.ui</td>
<td>3.9.2</td>
<td>value:</td>
<td>org.eclipse.jdt.ui.edit.text.java.organize.imports</td>
</tr>
<tr>
<td>executed</td>
<td>command</td>
<td>org.eclipse.ui.file.save</td>
<td>3.105.0</td>
<td>value:</td>
<td>org.eclipse.ui.file.save</td>
</tr>
</tbody>
</table>


References


Marko Gasparic is a Computer Science PhD Student at the Free University of Bozen-Bolzano, Italy. His research interests lie in recommendation systems for software engineering with a particular interest in improving the productivity of software developers by recommending useful IDE commands. He holds a master degree in computer science from the University of Maribor, Slovenia.

Gail C. Murphy is a Professor in the Department of Computer Science and Associate Dean (Research & Graduate Studies) at the University of British Columbia. Her research interests are in software developer and knowledge worker productivity. She holds a Ph.D. and M.S. degree from the University of Washington and a B.Sc. (Honours) degree from the University of Alberta.

Francesco Ricci is a professor of computer science at the Free University of Bozen-Bolzano, Italy. His current research interests include recommender systems, intelligent interfaces, mobile systems, machine learning, case-based reasoning, and the applications of ICT to health and tourism. He has published more than one hundred of academic papers on these topics. He is the editor in chief of the Journal of Information Technology & Tourism and on the editorial board of User Modeling and User Adapted Interaction.