Towards Persuasive Technologies for Improved Software Quality

Sophie Steinparz¹, Alexander Felfernig¹, Ingo Pribik², and Gerhard Leitner³

¹Institute for Software Technology, Applied Software Engineering, Graz University of Technology, Austria
²Flextronics, Corporate Software Development, Austria
³Institute for Informatics Systems, Interactive Systems, University of Klagenfurt, Austria

Abstract. Complexity metrics and refactoring rules have been developed with the goal to estimate the quality of software artefacts and to systematically improve their quality. There is still a significant gap between the prevalent quality of software artefacts and the potential quality which could be achieved if software developers would adopt well-known software development practices represented, for example, in complexity metrics and refactoring rules. The major goal of our work is to develop persuasive mechanisms that pro-actively motivate software developers to improve the overall quality of software artefacts. We have implemented these mechanisms as a plug-in for the software development environment Eclipse. The major goal of this short paper is to introduce the idea of persuasive technologies for improved software quality, to report the current status of our work, and to discuss issues for future research.

Key words: Persuasive Technologies, Motivational Systems, Software Engineering, Software Quality.

1 Introduction

High-level software quality is a major precondition for guaranteeing maintainability and reusability of software artefacts. Software complexity metrics [3] help to estimate the quality of a given set of software artefacts and to focus quality assurance activities. In addition to complexity metrics, refactoring rules [7] include detailed measures for improving the quality of a given set of artefacts.

Although complexity metrics and refactoring rules have already been integrated in software development environments [1], their application is very often restricted to non-commercial settings. A major issue in this context is that developers often ignore the availability of complexity metrics and refactoring support [12]. They rather trust in their ability to produce high quality artefacts. This attitude is misleading in the sense that programmers simply ignore the availability of supportive quality assurance tools.

Persuasive technology is defined as technology that is designed to change attitudes or behaviors of the users through persuasion and social influence, but
not through coercion [4–6]. There are several application scenarios for persuasive technologies, for example, improving energy conservation at home [10] or switching to a more healthy cooking behavior [2]. The major goal of our work is to develop persuasive technologies applicable within the context of software engineering scenarios. These technologies should motivate software developers to improve the quality of their artefacts and thus contribute to an increased reusability and maintainability. This goal will be achieved by identifying low-quality artefacts and to motivate software developers to improve these artefacts.

The remainder of this paper is organized as follows. In Section 2 we provide an overview of work in progress. We present user interface prototypes that have already been integrated into the Eclipse (www.eclipse.org) software development environment. In Section 3 we provide an overview of related work and give an outlook to future work. With Section 4 we conclude the paper.

2 Work in Progress

The first task in our work was to figure out which persuasive elements could be successfully applied to achieve persuasive effects in different software development scenarios. For the first version of our prototype environment we decided to implement the concepts of tunneling, suggestion, conditioning, and self-monitoring that have been introduced in [6].

Tunneling is a persuasion type [6] where users are guided through a predefined sequence of actions or events, step by step. As a realization of tunneling, we have implemented a quality guide which goes through a set of problematic software artefacts and provides explanations as to why these artefacts are problematic and which actions should be taken into account in order to improve the quality of the artefacts. A simple example of a quality guide supporting the tunneling principle is depicted in Figure 1.

![Fig. 1. Quality guide on the basis of the tunneling principle [6].](image-url)
Suggestion is a persuasive technology which means *intervening at the right time* [6], i.e., the system should provide advice regarding software artefact quality improvements at the right time. In our current implementation, we decided to implement the suggestive element on the basis of parts of the quality guide. Whenever a software developer is working on an artefact for a longer time and the quality of the artefact becomes worse or does not improve, the system will suggest improvement measures (e.g., link to the relevant parts of the quality guide) on the basis of quality metrics and refactoring rules.

Self-Monitoring is a persuasive technology which *allows people to monitor themselves to modify their attitudes or behaviors to achieve a predetermined goal or outcome* [6]. In our context we provide a couple of feedback icons that are constantly shown to the user in different areas of the Eclipse environment. The *package explorer* includes an overview of the quality of the software artefacts (see Figure 2). Immediate information regarding low-quality software artefacts is provided in the *Eclipse marker view* (see Figure 3).

Conditioning technology is a computerized system that uses principles of *operant conditioning to change behaviors* [6]. In this context, operant conditioning is a method that is based on the exploitation of positive reinforcement [6]. Currently, we have identified the possibility of exploiting different types of feedback icons to achieve such positive reinforcements. For example, every time the quality of a certain software artefact has been improved from one build to the next, the system will show a motivating image – currently, we display an image which summarizes the positive change since the last build.
To ensure usability of the presented interface elements, we have conducted an empirical study (structured interview) with computer science students at the Graz University of Technology (n=20 subjects participated in the study, 60 percent male and 40 percent female). The goal of this study was to identify a representative set of feedback icons – these were selected by the subjects of the study (see, for example, Figure 2). Furthermore, we were interested in how to best position the feedback information within the Eclipse development environment. Again, we evaluated the feedback of the study participants and positioned the feedback information conform to the articulated preferences (see Figure 4).\footnote{A detailed discussion of this user study has been omitted due to space limitations.} With another 10 participants we started an initial evaluation of our new software engineering environment. The results of the corresponding
interviews are summarized in Table 1. The current status of our work is that we have integrated our ideas in the form of a plugin for the Eclipse programming environment. Related and future work will both be discussed in the next section.

<table>
<thead>
<tr>
<th>ID</th>
<th>question</th>
<th>answer (examples)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>Why motivation for quality code?</td>
<td>concentration++, positive feedback</td>
</tr>
<tr>
<td>Q2</td>
<td>Why use it in your own work?</td>
<td>productivity++, quality awareness++</td>
</tr>
<tr>
<td>Q3</td>
<td>Exist advantages?</td>
<td>usability++, productivity++, errors–</td>
</tr>
<tr>
<td>Q4</td>
<td>Which organisational preconditions?</td>
<td>management support, understandable metrics</td>
</tr>
<tr>
<td>Q5</td>
<td>Which are desired functionalities?</td>
<td>shaded areas, personal code-meter</td>
</tr>
</tbody>
</table>

Table 1. Feedback of study participants.

3 Related and Future Work

Software complexity metrics have a long tradition in software engineering and have been applied in many different application contexts [3]. Typically, complexity metrics are represented by a set of rules that specify how to estimate the complexity of software artefacts. Basic metrics are simply related to the lines of code (LoC) or the number of used operands or operators. More complex metrics already take into account the structural complexity of software – one example for such a structural complexity metric is the cyclomatic complexity introduced by [11]. Since the work of [11], many additional complexity metrics have been developed which, for example, take into account the specific properties of object-oriented software [9]. Our work is independent of specific metrics but rather applicable with different types of complexity metrics.

Refactoring rules [7] play a major role in pro-actively supporting users in improving the quality of software artefacts. Rules are related to the overall complexity of a class, to the overall complexity of an inheritance hierarchy, or to the usage of nested if-statements (which could potentially be replaced by a more simple switch-statement). An example for the integration of refactoring rules into a development environment is given in [1]. This environment contains a so-called style check functionality which makes it easy for programmers (beginners) to learn more about how to develop high-quality software artefacts. Our environment does not rely on a specific set of refactoring rules – the concepts are generally applicable. What has to be changed depending on the used set of refactoring rules are the corresponding improvement instructions.

Persuasive technologies [6,8] can be found in a number of different application contexts. Example scenarios are the improvement of energy conservation at home (see, e.g., [10]) or the switching to a more healthy cooking behavior (see, e.g., [2]). Persuasive technologies play an important role in many different application domains [6] – this paradigm triggers the development of software components that pro-actively help to achieve a certain intended behavior. In the context of our work we are focusing on the development of persuasive technologies for software engineering scenarios which is a new domain for the development and application of persuasive technologies [12].
The work presented in this paper is a first summarization of ongoing work. The major focus of future work will be a couple of empirical studies with the goal to figure out the most effective user interface for triggering persuasive effects. Furthermore, we will compare the performance of persuasive interfaces with those of standard Eclipse interfaces in terms of output quality. We are also interested in the development of a standard Eclipse plugin that supports improved software development practices on the basis of persuasive technologies.

4 Conclusions

The major goal of our work is to develop persuasive technologies applicable within the context of software engineering. These technologies will motivate software developers to improve the quality of their artefacts and thus contribute to an increased reusability and maintainability. With our approach we are focusing on the active prevention of ”smelly” software artefacts on the basis of the psychological concept of persuasion. This way we introduce an orthogonal view on software quality improvement.

Acknowledgements. We want to thank the reviewers for their comments.

References


