Science of Computer Programming ••• (••••) •••–•••

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Continuous quality assessment with inCode

George Ganea, Ioana Verebi *, Radu Marinescu

Universitatea Politehnica Timişoara, Bvd. V. Parvan 2, Timişoara, Romania

ARTICLE INFO

Article history:
Received 5 March 2014
Received in revised form 16 December 2014
Accepted 23 February 2015
Available online xxxx

Keywords:
Quality assessment
Design quality
Design problems
Code smells
Software metrics

ABSTRACT

In spite of the progress that has been made over the last ten years in the research fields of software evolution and quality assessment, developers still do not take full advantage of the benefits of new assessment techniques that have been proposed by researchers. Beyond social factors, we believe that there are at least two main elements that contribute to this lack of adoption: (i) the insufficient integration of existing techniques in mainstream IDEs and (ii) the lack of support for a continuous (daily) usage of QA tools. In this context this paper introduces inCode, an Eclipse plugin aimed at transforming quality assessment and code inspections from a standalone activity, into a continuous process, fully integrated in the development life-cycle. But inCode not only assesses continuously the quality of Java systems; it also assists developers in taking restructuring decisions, and even supports them in triggering non-standard, complex refactorings. This paper introduces inCode's differentiating features, it presents the design goals that shaped our construction decisions, and describes a controlled experiment designed to validate the usability of the tool. The experiment has indicated that developers using inCode are more efficient in refactoring design fragments affected by design flaws.

1. Introduction

There is no perfect software design. Like all human activities, the process of designing software is error prone; and object-oriented design is no exception. The flaws of the design structure, also known as "bad smells" [11], increase change- and fault-proneness in code, which in turn leads to a sensible increase of maintenance costs [17]. Thus, the detection and correction of design flaws is essential for improving software quality.

All of today's major IDEs provide extensive quality assurance modules that use metrics to control design quality. However, using metrics in isolation makes it very hard to assess properly a design aspect (*e.g.*, the distribution of system's intelligence among classes) [23]. The problem is that individual measurements may indicate a code anomaly, but they usually leave the engineer clueless concerning the *higher-level cause* of the anomaly. Consequently, it is very hard to infer reasonable restructuring measures based strictly on some abnormal metric values.

The typical usage scenario of a quality assessment tool is currently this: a developer, *feeling* that something is wrong with her code, uses the QA tool provided by her IDE to compute a suite of metrics; noticing some abnormal metric values, she must *infer* the *real* design problem by interpreting a set of metric values. This is not easy, especially if this assessment occurs *long after* that code/design fragment has been created, or if the code was written by other developers. Moreover,

* Corresponding author.
E-mail addresses: georgeganea@gmail.com (G. Ganea), ioanaverebi@gmail.com (I. Verebi), radu.marinescu@cs.upt.ro (R. Marinescu).

http://dx.doi.org/10.1016/j.scico.2015.02.007
0167-6423/

after grasping the real design problem, correction is still a challenging and painstaking task: the developer must find a way to compose a proper restructuring solution from the basic refactoring operations available in the IDE.

In this paper we introduce inCode,[1] an Eclipse plugin aimed at transforming quality assessment and code inspections from a standalone activity, into a continuous process, fully integrated in the development lifecycle. inCode supports programmers with continuous detection of design and code problems (*e.g.*, duplicated code [11], classes that break encapsulation [34], or misplaced methods). inCode identifies and locates specific design flaws *as they appear*, and it guides developers in correcting the detected flaws, by providing them with *contextualized explanations* for each instance. This improves the efficiency of performing code reviews, because it allows developers to address design problems earlier *i.e.*, when the entire design context is still fresh to them, and because it provides them with hints that may lead to a better refactoring solution.

The efficiency improvement brought by continuous design quality analysis over periodic code reviews remains to be extensively evaluated. However, the controlled experiment presented in this paper brings some encouraging evidence that developers using the contextualized information provided by inCode are more efficient in refactoring code fragments affected by design problems. The adoption of a new quality assessment approach by the industry is certainly a complex matter. Telea et al. [40] quote senior managers stating that adoption depends on the *measurable* added value brought by a tool and the corresponding cost. Bessey et al. [1] indicate that adoption might be also challenged by the diversity of language dialects, compilers and platforms used, and by social restrictions (dis)allowing certain modifications. Kemerer [16] also emphasizes that the learning curve plays a important role in a tool adoption. Moreover, in order to be truly efficient, continuous assessment demands not only adequate tools, but also a clear process (methodology). While we are aware of all these issues, the main focus of this paper is to show that, from a technical point of view, tools assessing quality continuously are suitable, even when used on large-scale projects.

The rest of the paper is organized as follows: Section 2 illustrates how inCode supports continuous quality assessment by means of a typical usage scenario that "interweaves" current development and quality assessment. inCode's operation mode and its main distinctive features are then summarized in Section 3. In Section 4 the focus is on inCode's construction: we state the three main design goals that shaped inCode and describe in detail how these goals are reflected in the concrete design decisions that we made. In the second part of the paper we evaluate inCode from two different perspectives: its usability and the extent to which the three design goals have been met. Thus, in Section 5 we describe a controlled experiment that was performed on ten subjects to assess if and how inCode can help developers become more efficient in dealing with design problems. Then, in Section 6 the three design goals are evaluated, including a detailed discussion of inCode's time and memory performance. The paper is wrapped-up by a discussion of the most important lessons learned (Section 7), a summary of related work (Section 8), and a series of conclusive remarks and future perspectives.

2. A typical scenario of continuous assessment

Before describing in more detail the features of inCode, we will illustrate by a simple yet suggestive example its main trait: the capacity to automate in a continuous manner the detection of design problems, as well as its context-sensitive support for automatic restructuring.

The scenario starts with Lisa, a typical developer, beginning to write the Rectangle class (see Fig. 1, **Step 1**)[2] in Eclipse's editor window. The very moment she finishes writing the 4 lines of the Rectangle class and saves the file, a red inCode *marker* appears on the left side ruler of her editor, next to the class definition. This notifies her that inCode has detected a potential design problem. In this example, the reader may notice that Rectangle is a class containing four *public* attributes, which potentially break encapsulation. This design problem is known in the literature as *Data Class* [11,34], and in the past metrics-based rules have been defined for automatically detecting such design problems [20]. So, every time Lisa changes the code, saving the file triggers inCode to execute its metrics-based detection rules and display a red marker for each detected design problem.

Noticing the red inCode marker, Lisa wants to find out what the problem is. inCode markers behave exactly like the standard Eclipse ones which signal compiler errors or warnings; therefore, it comes naturally to Lisa to click on it (**Step 2**) and find out that a *Data Class* problem has been detected.

Next, when Lisa decides to find out more about the problem, the inCode Tips view is opened and she reads a detailed description of what the problem is. One important thing here is that the description is not a presentation of what *Data Class* means in *general*, but an explanation of why Rectangle, *in particular*, is reported by inCode as a *Data Class* (see text box next to **Step 2**). inCode Tips also includes a section of *Refactoring Tips*, which in this particular case advises Lisa to encapsulate the four attributes of Rectangle because they have no reason to be declared public, as no one is using them from outside the class. Again, the noteworthy aspect here is the *context-sensitive nature* of the refactoring advice.

Finally, note that, in order to use inCode, Lisa is not required to have an understanding of any of the many metrics on which the detection of the problem is based on. Lisa does not even need to know exactly what a *Data Class* is. We believe that this is an essential trait for any quality assurance tool that aims for a wide adoption by developers: it has to hide the

[1] inCode can be installed using the standard Eclipse mechanism, using the following update site: http://www.intooitus.com/research/incode/.

[2] The sequence of steps in our example are summarized in form of numerical labels (which we are going to refer to in the following as **Step X**).

Please cite this article in press as: G. Ganea et al., Continuous quality assessment with inCode, Sci. Comput. Program. (2015), http://dx.doi.org/10.1016/j.scico.2015.02.007