Lightweight Visualizations for Inspecting Code Smells

Chris Parnin, Carsten Görg*
College of Computing
Georgia Institute of Technology
Atlanta, Georgia
{vector,goerg}@cc.gatech.edu

Abstract

In this paper we present an approach for using lightweight visualizations to inspect bad coding patterns. We demonstrate how our Visual Studio plugin assists developers in finding relevant methods to inspect.

CR Categories: D.2.9 [Software Engineering]: Management—Software quality assurance;

Keywords: quality assurance, code smells, software visualization

1 Introduction

Quality assurance of source code is a vital process in regulating the future costs of a software product. Poorly structured code harbors future bugs, is difficult to understand, and is resistant to change. Software inspection is a technique that developers employ to manually review the quality of source code. To perform an inspection, developers use intuition guided by past exposure to problematic structures and common design mistakes to uncover flaws in the source code. To effectively reduce the long-term cost of developing source code, developers need to easily identify problematic source code and eliminate these defects early in the development process.

Several tools have been proposed to assist developers with inspecting the quality of source code. These tools typically use a set of rules to critique the source code. These rules are often seeded from object-oriented guidelines [Riel 1996] or code smells [Fowler et al. 2001]. After running a tool, the developer is presented with a ranked list of troublesome methods that have violated the rules in some manner. Other systems such as `jCOSMO` [Emden and Moonen 2002] present the results in a graph visualization of the source code. This system allows the developer to interactively explore code elements having code smells.

We believe these systems are not designed in alignment with how software inspection is performed in practice. The systems are designed for the developer to inspect and explore the defects of the entire system. Having the developer investigate many types and instances of problems is an overwhelming task. In addition, the design assumes that the most problematic code element is also the most interesting. In practice, developers do not always have a clear time allocated toward solely performing software inspection; instead, inspection is often interleaved with other software development activities. In discussions with professional developers, a common theme emerged: frustration with code critic systems generating too many results. The developers could not easily distinguish relevant problems because there were too many problems to tackle at once with many instances outside their scope of concern. For example, stable code sometimes had many design flaws flagged but was too costly to change; instead, developers wanted suggestions to be related to recent modifications of modules.

2 Approach

We designed a software inspection tool that limits the time needed to investigate source code candidates that may need to be refactored. The intent of the design is to allow the developer to interleave normal development tasks with a lightweight software inspection process.

A lightweight inspection process requires less exploration and the ability to focus on relevant elements. In this design, developers do not start from an overview of the entire source code and then drill down. In addition, developers do not encounter a multitude of possible defect types. Instead, developers will engage in task-directed searches that are relevant to their current context.

The visualization of the results of a task-directed query is designed so that the developer may eliminate false positives without manual inspection. A usage context of elements that the developer has recently expressed interest in is built using the technique presented in [Parnin and Görg 2006]. Query result candidates that are within the usage context are emphasized to allow the developer to examine elements relevant to their current task.

3 Application

We developed a Visual Studio plugin to demonstrate our approach. The tool allows the developer to search for various "bad smells" [Fowler et al. 2001] in source code. One such smell is, "long method", an indication to refactor the elements into smaller units in order to reduce code duplication and improve code understanding.

To perform a search for source code having a "long method" smell, the developer invokes the search plugin. The developer is then presented with a result set of candidate methods that are sorted by length. Within this result set, several false positives may exist. For instance, auto-generated user interface code and unit tests often contain many statements thus appearing high in rank.

The developer needs to distinguish plausible candidates for refactoring from false positives without manual inspection of the source code. Fowler et al. [2001] suggest additional tell-tale signs to investigate when searching for "long method" smells. In particular,

*The author was supported by a fellowship within the Postdoc Program of the German Academic Exchange Service (DAAD).

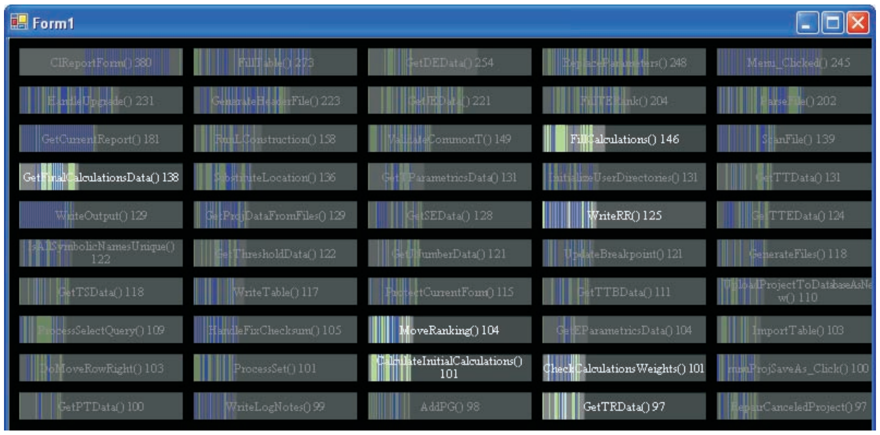

Figure 4: Displaying task-relevant candidates.

the presence of conditionals, loops, and comments are given as further reason to investigate code. The presence of conditionals and loops is clear-cut; too much complexity in one method results in a method that is difficult to modify and understand. The rationale for investigating a method with many comments in the body is more subjective. In some cases, comments explain complexity within the method that may be necessary to rewrite. Additionally, comments are often used to summarize the purpose of a sequence of statements; alternatively, these statements can be extracted into a new method named after the comment summary.

3.1 Method Views

In this section, we provide examples of several views and the insights they offer. The views are line-based visualizations of the source code methods. The lines are drawn vertically to provide room to display the method name. Light-blue lines are conditionals, blue lines are iteration statements, and green lines are comments. The method name and line count is displayed as white text. The views are scaled to be relative to the longest method – the light gray bar in the background indicates the length of the given method.

When inspecting code, a developer wants to avoid false positives. The first example (see Figure 1) is an auto-generated method for building an user interface. The visualization indicates that no complex statements are present which allows the developer to avoid further investigation.

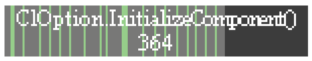

Figure 1: Auto-generated code.

In the second example (see Figure 2), iteration statements compose a large percentage of the method. Upon further inspection, the developer can see that the complex logic needs to be refactored.

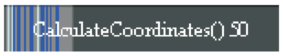

Figure 2: Complex method body.

In the third example (see Figure 3), the method has sections delimited by comments. The method is composed of several single line comments that summarize the functionality of the following statements. Upon further inspection, the developer observes that each section can be refactored into a new method.

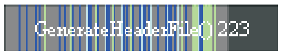

Figure 3: Comment summary of statements.

3.2 Task-relevant Inspection

In this example (see Figure 4), a developer was investigating changes to calculation code. After making some initial changes, the developer wanted to assess the overall quality of the code concerned with calculation in order to see if refactoring was necessary.

Using the search tool, the developer queried the code containing "long methods". However, unrelated code of worse quality flood the results. To alleviate this problem, the developer can filter the results to include the methods that were recently interacted with in the IDE. This allows the developer to view all the methods related to the current task while being able to relate the results in context with the rest of the program.

References

EMDEN, E. V., AND MOONEN, L. 2002. Java quality assurance by detecting code smells. In *WCRE '02: Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)*, IEEE Computer Society, Washington, DC, USA, 97.

FOWLER, M., BECK, K., BRANT, J., OPDYKE, W., AND ROBERTS, D. 2001. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

PARNIN, C., AND GÖRG, C. 2006. Building usage contexts during program comprehension. In *ICPC '06: Proceedings of the 14th International Conference on Program Comprehension*, IEEE Computer Society, Washington, DC, USA.

RIEL, A. J. 1996. *Object-Oriented Design Heuristics*. Addison-Wesley.