

Scalable, Expressive, and Context-Sensitive Code Smell Display

Emerson Murphy-Hill

Portland State University
emerson@cs.pdx.edu

Abstract

Code smell detectors can potentially help programmers identify opportunities to improve the design of software through refactoring. Unfortunately, the user interfaces to existing detectors often do not align with how programmers typically refactor. I argue the importance of scalability, expressivity, and context-sensitivity when displaying smells, and present a prototype tool that embodies these properties.

Categories and Subject Descriptors D.2.3 [Software Engineering]: Coding Tools and Techniques

General Terms Design, Human Factors

Keywords smells, refactoring, tools

1. Background and Problem

The term *code smell* was popularized by Martin Fowler in his book on *refactoring* [3]. Refactoring changes the design of a program without changing its behavior in order to help programmers understand code, add new features, and fix bugs. Fowler proposed 22 code smells that help programmers recognize candidates for refactoring, including LARGE METHOD, LONG PARAMETER LIST, and DATA CLUMPS.

A smell detector is a tool that automatically finds code smells and presents them to the programmer. A smell detector is useful because recognizing smells is a difficult and time consuming task. Refactoring frequently occurs continuously during development [5], and thus manually recognizing smells burden programmers with having to continuously determine whether any of the 22 smells are present as they navigate and edit code. Moreover, certain code smells can be difficult for programmers to spot manually [4].

2. Related Work

Several smell detectors have been implemented for a handful of programming languages. A common approach, imple-

mented both as research prototypes [1, 8, 9] and in development environments such as Eclipse (<http://eclipse.org>) and IDEA (<http://jetbrains.com/idea>), is to display a smell by highlighting program text related to an instance of a smell. Unfortunately, this approach *scales* poorly when displaying code smells. For instance, COMMENTS, MESSAGE CHAINS, and DUPLICATED CODE can occur with great frequency and also overlap the same location in program text, and so highlighting every occurrence of a smell may be overwhelming. Thus, a scalable smell detector is one that displays smell information without overwhelming the programmer.

Moreover, while the highlighting approach may capture the “where” of a code smell, it is not necessarily sufficiently *expressive* enough to convey the “why” of a code smell. For instance, suppose the a smell detector highlights 10 snippets of DUPLICATED CODE. Highlighting fails to communicate which snippets are duplicates of which other snippets, how the snippets differ (if at all), and whether any code outside the current editor is a duplicate of the highlighted snippets. Thus, an expressive smell detector is one that clearly explains why detected smells are present in code.

Some research prototypes have demonstrated visualizations of code smells [2, 6, 7]. However, these visualizations tend to separate the task of finding code smells from the task of refactoring by making the smell detector distinct from the program editor where refactoring typically occurs. While these visualizations may be useful for high-level code inspections, they may not be sufficiently integrated to be useful during frequent, continuous refactoring [5], where only smells that are relevant to the current programming *context* are useful. Thus, a context-sensitive smell detector is one that emphasizes smells within the current programming context over smells outside of it.

3. A Novel Approach to Code Smell Display

To demonstrate how scalability, expressivity, and context-sensitivity can be used in a smell detection tool, I have built a prototype tool as a plugin to the Eclipse Java development environment. While the following description should convey a rough idea how the tool works, a short screencast can be found at <http://multiview.cs.pdx.edu/refactoring/smells>.

When the programmer is browsing and editing source code, the smell detector shows a half-circle consisting of

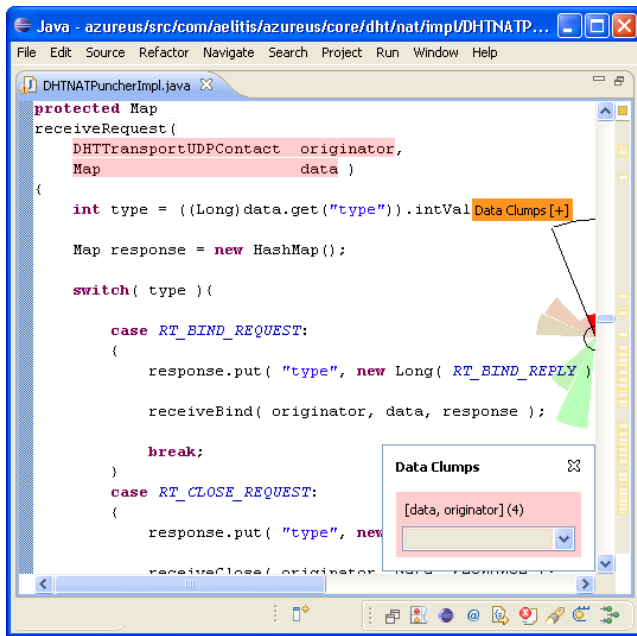


Figure 1. A screenshot of the code smell detector in use. Example code from Vuze (<http://vuze.com>).

wedges in the program editor (Fig. 1, center-right). Each wedge represents a smell, and the radius of the wedge is an estimate of the strength of the smell in the programmer’s current working context. When a wedge is moused-over, a label reveals the name of the associated smell (Fig. 1, “Data Clumps [+]) and the wedge’s color and edges are accentuated. This visualization was built to provide an always-available, at-a-glance representation of the context-sensitive code smells in a scalable manner.

When the programmer wants more detail about a particular smell, she may click on a smell label to reveal a more expressive view of the sources of the smell in the current context. Although this expressive view differs from smell to smell, the visualizations for the 8 smells that we have implemented so far have two components in common: editor overlays and a summary window. To take an example, the summary window in the lower right of the editor in Figure 1 shows that for the DATA CLUMP smell, two variables (data and originator) appear together in 4 locations. Choosing a method name from the combo box (Fig. 1, below “[data,originator](4)”) navigates to each of the 4 instances. An instance of the data clump is highlighted by a pink editor overlay, and is colored so that the programmer can visually associate the overlays with each other and with the summary window. More complicated smells require more complicated explanations, but the tool is both scalable and expressive because detailed smell explanations are shown only on demand.

4. Contributions

This research makes two primary contributions:

1. The identification and motivation of three properties for code smell display: scalability, expressivity, and context-sensitivity.
2. An visualization that exhibits these properties in the form of a prototype tool that is highly-integrated.

Furthermore, I expect to make a third contribution in future work:

3. An evaluation of the prototype to determine whether the tool, and the properties it embodies, produces the supposed benefits of smell detectors.

Acknowledgments

Thanks to my advisor, Andrew P. Black, and the National Science Foundation for funding under CCF-0520346.

References

- [1] M. Bisanz. Pattern-based smell detection in TTCN-3 test suites. Master’s thesis, Georg-August-Universität Göttingen, Dec. 2006.
- [2] E. van Emden and L. Moonen. Java quality assurance by detecting code smells. In *Proceedings of the Ninth Working Conference on Reverse Engineering*, pages 97–106, Washington, DC, USA, 2002. IEEE Computer Society.
- [3] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
- [4] M. V. Mäntylä. An experiment on subjective evolvability evaluation of object-oriented software: explaining factors and interrater agreement. In *Proceedings of the International Symposium on Empirical Software Engineering*, pages 287–296, Nov. 2005.
- [5] E. Murphy-Hill and A. P. Black. Why don’t people use refactoring tools? In *Proceedings of the 1st Workshop on Refactoring Tools*, pages 61–62, Berlin, Germany, 2007.
- [6] C. Parnin and C. Görg. A catalogue of lightweight visualizations to support code smell inspection. In *Proceedings of the 2008 ACM Symposium on Software Visualization*, 2008.
- [7] F. Simon, F. Steinbrückner, and C. Lewerentz. Metrics based refactoring. In *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering*, pages 30–38, Washington, DC, USA, 2001. IEEE Computer Society.
- [8] S. Slinger. Code smell detection in Eclipse. Master’s thesis, Delft University of Technology, Mar. 2005.
- [9] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou. Jdeodorant: Identification and removal of type-checking bad smells. In *Proceedings of the 12th European Conference on Software Maintenance and Reengineering*, pages 329–331. IEEE Computer Society, 2008.