Seven Habits of a Highly Effective Smell Detector

Emerson Murphy-Hill and Andrew P. Black
Portland State University
emerson@cs.pdx.edu, black@cs.pdx.edu

ABSTRACT

The process of refactoring code — changing its structure while preserving its meaning — has been identified as an important way of maintaining code quality over time. However, it is sometimes difficult for progammers to identify which pieces of code are in need of refactoring. "Smell detectors" are designed to help programmers in this task, but most smell detectors not mesh well with "floss refactoring," the common tactic in which refactoring and programming are finely interleaved. In this paper we present a smell detector that we have built with floss refactoring in mind by combining seven principles, or habits, for designing usable smell detectors. We hope that this combination can help the designers of future smell detectors build tools that align with the way that programmers refactor.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques ; D.2.6 [**Software Engineering**]: Programming Environments

General Terms

Design, Human Factors

Keywords

smells, refactoring, tools

1. INTRODUCTION

Refactoring is the practice of restructuring code without changing its externally observable behavior. In his influential book, Fowler advocates frequent refactoring because it helps programmers understand code, find bugs, and add new features [4]. These recommendations do indeed seem to produce benefits in practice; several studies have shown quantitative improvements to code bases as a result of refactoring [7, 5, 9] and others have confirmed that it is a frequent practice [17, 18]. To automate the process of transforming code, refactoring tools have been introduced for several environments, including IntelliJ IDEA (http://www.jetbrains.com/idea/), Omnicore X-Develop (http://www.omnicore.com/en/xdevelop.htm), and Xcode (http://developer.apple.com/tools/xcode/).

But before a programmer can refactor code, she must first recognize code that needs to be refactored. For example, consider the following code snippet:

```
class TrainStation{
  int lengthOf(Train t) {
    return  t.locomotiveCount() +
            t.boxcarCount() +
            1; //the caboose
  }
...
```

The method `lengthOf` exhibits the FEATURE ENVY smell, because the method sends several messages to a `Train` object, but sends no messages to itself. FEATURE ENVY is a problem that can make software more difficult to change because a class's responsibilities are contained not only in the class itself, but also spread throughout envious classes that access the class's members. Table 1 describes several other code smells.

This smell can be alleviated by delegating the functionality to the `Train` class:

```
class TrainStation{
  int lengthOf(Train t) {
    return  t.length();
  }
...
class Train{
  int length() {
    return  locomotiveCount() +
            boxcarCount() +
            1; //the caboose
  }
...
```

This particular refactoring can be achieved by sequencing three smaller refactorings: Extract Method, Move Method, and Rename Method. But the mechanics of these refactorings are not the topic of this paper, which instead focuses on *how* the programmer recognized that this code needs to be refactored.

Until recently, programmers have been forced to locate smells manually. Because finding smells comes directly before refactoring, the task of finding smells occurs during one of three other programming activities: adding features, fixing bugs, or doing a code review [4, pp. 58–59]. However, looking for smells during these activities can be difficult. There are two reasons for this. First, novice programmers sometimes cannot locate smells as proficiently as more experienced programmers, as Mäntylä has shown in an experiment [11]. Second, because of the number of smells (22 listed

DATA CLUMPS	A group of data objects that are duplicated across code [4]
FEATURE ENVY	Code that uses many features from classes other than its own [4]
REFUSED BEQUEST	A method that overrides a superclass method, but does not use the super method's functionality [4]
SWITCH STATEMENT	A switch statement, typically duplicated across code [4]
MESSAGE CHAIN	A series of method calls to "drill down" to a desired object [4]
TYPECAST	Changing an object from a one type to another type [3]
INSTANCEOF	An operator that introspects on the type of an object [3]
MAGIC NUMBER	A hard-coded value that is poorly documented [4]
LONG METHOD	A method with too much code [4]
LARGE CLASS	A class with too much code [4]
COMMENTS	Comments denote code that is not self-explanatory [4]

Table 1: Some smell names and descriptions

in Fowler's book alone [4]), it is impractical to expect a programmer, even an expert programmer, to look for all smells at all times.

Fortunately, many smells can be detected automatically by tools. One of the first smell detection tools provided code smell visualizations to help programmers locate code smells. In Van Emden and Moonen's jCosmo [3], the tool analyzes the *entire program* and displays a graph; the size and color of the graph nodes show which parts of the system are affected by which code smells.

As we have suggested previously [12], it appears that programmers refactor frequently to maintain healthy code, interleaving refactoring with other tasks such as adding features. We call this *floss refactoring*, and contrast it with *root canal refactoring*, where a programmer refactors code intensively and exclusively once it has become unhealthy.

If a programmer were to use a smell detector during floss refactoring, she would need to run it frequently, interleaved with program modifications. Because the programmer performs floss refactorings only if they help accomplish an immediate programming goal, the programmer would not be interested in smells in code unrelated to the code currently being worked on. However, because tools like jCosmo display smells for the entire program, and thus take considerable time to analyze the code, these tools are not appropriate for smell detection during floss refactoring.

CodeNose [15] addresses the limitations of jCosmo by presenting programmers with a representation of detected smells inside the editor. Built on top of the Eclipse programming environment (http://www.eclipse.org/), CodeNose underlines locations in the program text where smells have been detected, much like Eclipse's standard compiler warnings:

```
(Class<Long>) Class.getPrimitiveClass("long");
```

A similar line-based indicator for smell detectors has been independently proposed by Hayashi and colleagues [6], Bisanz [1], and Tsantalis and colleagues [16]. Indeed, the presentation of smells in the same manner as compiler warnings is intuitively appealing for several reasons. First, both smells and warnings tell the programmer about a "problem" with the source code, so it makes sense to report them similarly. Second, underlining is the default user interface mechanism supported by static analysis tools such as the Eclipse Test and Performance Tools Platform (http://www.eclipse.org/tptp/), where programmers can easily build their own custom smell detection tools. Third, underlining allows the programmer to quickly see smells while editing code, making the presentation apparently suitable for floss refactoring.

However, we argue that two characteristics of code smells make underlining an inappropriate mechanisms for communicating the smells to the programmer:

1. Whereas a piece of code either generates a compiler warning or it does not, a code smell may be subtle or flagrant, with many shades between. For example, whether a method has the LONG METHOD smell depends on what the programmer considers "too long", which may depend on the context of the program. Smell detectors that underline code typically deal with this by using thresholds, but static thresholds may not be sufficiently flexible in every context.

2. Using underlining for code smell identification does not scale well. Consider again the `TrainStation` example above. While it is a relatively small method, it contains at least three code smells: FEATURE ENVY, MAGIC NUMBER, and COMMENTS. Depending on the rest of the program, other smells like REFUSED BEQUEST may be present in the snippet as well. Indeed, nearly *all* the code in this method smells! Underlining everything that contains even a whiff of a code smell could quickly overwhelm the programmer, making the detector useless.

We see that presenting detected code smells is difficult because of their complexity and abundance. Previous researchers have worked towards explaining smell complexity by providing expressive visualizations of code smells [14, 13], but do not discuss how to manage smell abundance. This paper addresses how a smell detector can handle both the complexity *and* the abundance of smells.

2. THE SEVEN HABITS

In this section, we identify seven characteristics of an effective smell detector, based on the nature of code smells themselves and the context in which programmers find them.

Scalability

Code smells can emanate from many pieces of code, and the same code can give off several smells. A smell detector should not only be able to explain the detected smells in detail, but should also do so without overloading the programmer.

Availability

Performing refactoring is often a frequent process during program development, and therefore finding code smells is also frequent. Rather than forcing the programmer to frequently go through a series of drawn out steps in order to see if a tool finds any code smells, a smell detector should make smell information as available as soon as possible, with no effort on the part of the programmer.

Unobtrusiveness

Because programmers refactor frequently during floss refactoring, finding smells is inherently intermingled with other kinds of pro-

gram edits: it is not a separate task. Thus, a smell detection tool should be unobtrusive and not block the programmer while the tool gathers, analyzes, and displays smell information.

Context-Sensitivity

Code smells may occur in any part of code base, but it is most important to fix those that are directly relevant to the current programming task. Indeed, fixing smells in a context-*insensitive* manner may be a premature optimization. Therefore, a smell detector should first and foremost point out smells relevant to the current programming context.

Expressiveness

Many code smells are inherently complex; they can relate to several program elements that may be distributed in many places across the program text. A smell detector should go further than simply telling the programmer that a smell exists; it should help the programmer find the source(s) of the problem by explaining *why* the smell exists.

Relativity

Not every kind of smell that a tool can detect has equal value to the programmer. This is because some smells are obvious to the naked eye (e.g., LONG METHOD), while others are difficult for a programmer to find (e.g., FEATURE ENVY) [11]. Thus, a tool should place more emphasis on the smells that are more difficult to recognize without a tool.

Relationality

Many smells do not emanate from a single point in the code, but instead speak about relationships between several program elements. Once again FEATURE ENVY is a good example: it is caused by a relationship between a class and its clients. Thus, a smell detection tool should be capable of showing relationships between code fragments that give rise to smells.

3. AN EFFECTIVE SMELL DETECTOR

We have built a prototype smell detector that gratifies the seven habits. The tool provides three different views on the code smells: it starts in Ambient View, shows an overview in Active View, and finally reveals smell details in Explanation View. While we describe the tool textually in this section, it is more useful to see it in action; a series of short screencasts can be found at http://multiview.cs.pdx.edu/refactoring/smells.

3.1 Ambient View

The initial view of the smell detector is ambient, where a visual representation of contextually relevant smells is displayed in the program editor, behind the program text (Fig. 1, top). This visualization is intended to be visible and **available** at all times during code editing and navigating, translucent enough as to be **unobtrusive**.

The visualization is composed of circular sectors, which we call wedges, radiating from a central point in a half-circle. Each wedge represents a code smell, and the radius of each wedge represents the degree to which the current programming context exhibits that smell. For example, at the top of Figure 1, the southernmost wedge indicates the strongest smell while the northernmost wedge indicates the weakest smell. As the programmer navigates through the code, the location of the wedges on screen remain fixed, but the radius of each of the wedges change as the programmer's context changes. The radius of each wedge is controlled by a smell analyzer that evaluates a smell in the programmer's **context**. However,

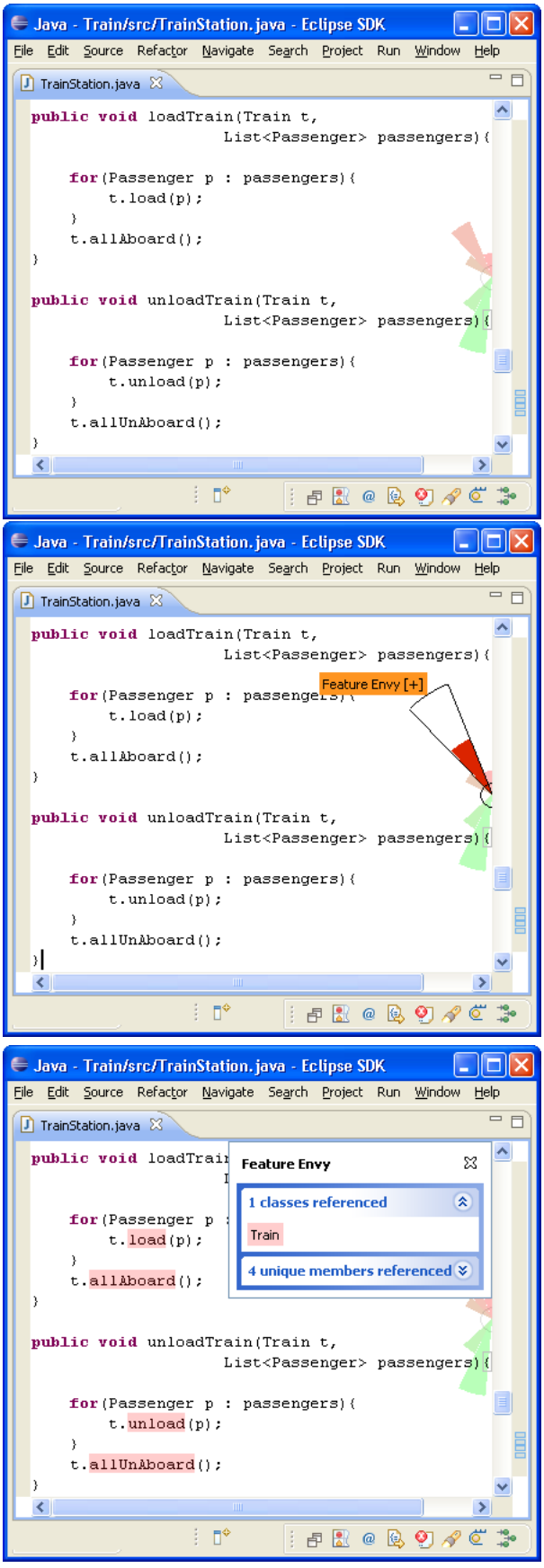

Figure 1: Three progressively more informative views of the code smells in program code.

the maximum screen area available for each wedge is bounded, and thus the visualization is designed to **scale** as the number of smells increases.

Wedges are colored from red-to-green, north-to-south. Smells are assigned to wedges such that the southernmost (and greenest) wedge is the most obvious smell and the northernmost (and reddest) wedge is the least obvious smell. Our subjective obviousness ordering is reflected in Table 1, where the least obvious smell appears at the top, as it does in Ambient View[1]. For example, at the top of Figure 1, the view indicates that there is a strong unobvious smell (in this case, FEATURE ENVY, although the smell names are intentionally omitted from this view) as well as a strong obvious smell (LARGE CLASS). Because more obvious smells are distinguished from less obvious smells, both spatially and chromatically, the programmer can judge the **relative** importance of the displayed smells.

The purpose of this view is to allow the programmer to occasionally glance at the visualization to determine if there are any strong, relevant code smells and to give a rough estimate of their extent. There is very little burden or commitment on the part of a programmer to determine whether a smell exists; she only needs to glance at the visualization, unlike with most existing smell detection tools which require the programmer to activate the tool and inspect the results. Indeed, such batch-smell-detection tools are applicable in only 1 of 3 refactoring tasks (code inspection) [4, pp.58-59], and are less useful for floss refactoring.

3.2 Active View

If the programmer observes something interesting or unusual in the Ambient View, she can then mouse-over a wedge to reveal the name of that wedge's associated smell. In the middle of Figure 1, we have moused-over the second smell from the top. If the programmer is interested in further details of the detected smell, she can click on the smell label to activate the Explanation View.

The purpose of the Active View is to provide a little more information than the Ambient View, and to help the programmer transition to the Explanation View. Again, the transition from view to view in order to reveal more information was designed to be as fast and as painless as possible.

3.3 Explanation View

The Explanation View provides detailed information about a particular smell. In essence, this view was designed to explain *why* the smell's wedge has the displayed radius in the most **expressive** way possible. Each smell is displayed using different visual elements, but smells in the Explanation View typically have two common components.

Summary Pane. In the bottom panel of Figure 1, a summary pane is displayed at the upper right of the editor. This pane is fixed relative to the program code (that is, it does not move when the programmer navigates away), but may be moved manually within the editor at the programmer's behest. Generally, this pane displays a summary of the data collected from the smell analyzer. In the Figure, the summary pane for the FEATURE ENVY smell shows that 4 members from the `Passenger` class are accessed.

Editor Annotations. The code editor is typically annotated to point out where smells originate from. For example, in the Figure you can see where the 4 members of the `Passenger` class are referenced in the editor. Each member reference to an external class is **related** together visually using the same color highlight.

[1]We currently have implemented analyzers for all the smells in Table 1, with the exception of REFUSED BEQUEST, MAGIC NUMBER, and COMMENTS.

Taken together, editor annotations and the summary pane were built to help the programmer not only understand *if* code smells, but *why* the code smells.

3.4 Technical Details

While we have outlined how our smell detection tool works in general, a number of technical details have significant bearing on the tool's practicality.

First, how does the tool determine the radius of each wedge in the Ambient View? The maximum size of each wedge is fixed, so that it does not monopolize the program editor. An individual smell analyzer is responsible for converting a smell in the programmer's working context to a scalar value between zero and the maximum radius of the wedge. While the formula for the radius is different for different analyzers, some formulas are more complex than others. For instance, LARGE CLASS is a relatively simple formula because the radius increases as the size of the class increases, while FEATURE ENVY incorporates the number of external classes referenced, the number of external members referenced, and whether internal members are referenced.

Second, how does the tool search for smells efficiently? Several smell analyzers require complex program analysis, so as the number and complexity of analyzers increase, the development environment may begin to respond more slowly. However, having detection run in a background thread and caching smell results for unchanged program elements have been important techniques for maintaining acceptable performance. Moreover, we hope that a more intelligent search strategy, starting in the programmer's current context and radiating outward to less contextually relevant code, will improve performance even further.

Third, what constitutes the programmer's "current context?" In our implementation, we define current context as the the union of all methods, whole or partial, that are visible in the open editor. More sophisticated definitions of context may be used as well, such as task contexts used by the Mylyn tool [8] or Parnin and Görg's usage contexts [**?**].

4. PLANNED EVALUATION

We plan on completing an evaluation in the near future. We are primarily interested in showing that our "habits," as embodied by the tool, are useful in helping programmers do their jobs. To that end, we propose an experiment with three parts.

In the first part, we ask the programmer to inspect several pieces of source code, and then tell us when the detector is displaying an interesting smell. We will also purposely tell the programmer to pause at specific points in the source code, and tell us which smells and in which order they would ask the tool for more detailed information. The purpose of this part of the experiment is to determine if the tool conveys smell information to the programmer efficiently.

In the second part of the experiment, we ask programmers to use the Explanation View to learn more about one or two smells. Each time they use the Explanation View, we will ask the programmer about the extent of the smell, how likely she is to correct the smell, and whether the display tells her something surprising about the code. The purpose of this part of the experiment is to determine whether the Explanation View is sufficiently detailed to convey complex smell information.

We will repeat these two parts with each programmer, except that we will not give the programmer the aid of the tool. Our intent is to establish a baseline for judging the programmers' behaviors, and to allow programmers to respond in a more comparative manner in the final part of the experiment.

In the final part, we ask programmers to subjectively evaluate

their experience with the tool. We will use a questionnaire where the programmers answer a variety of questions about whether the tool helped them quickly and efficiently find and understand smells in the code. Moreover, we will ask a series of questions regarding how the programmers might use the tool in their own code.

We hope that this proposed evaluation will help show that our smell detection tool, and thus our guidelines, are an effective approach to smell detection.

5. CONCLUSION

We have presented seven principles for the design of code smell detectors that fit into the typical floss refactoring workflow, and described a realization of these principles in the form of an ambient smell detector. While we feel that the seven principles are important to building usable smell detectors, we have not yet verified that this is the case, and plan to do so in a forthcoming experiment.

6. ACKNOWLEDGEMENTS

Thanks to Chris Parnin and Claudia Rocha for their helpful advice, and to the National Science Foundation for partially funding this research under CCF-0520346.

7. REFERENCES

[1] M. Bisanz. Pattern-based smell detection in TTCN-3 test suites. Master's thesis, Masterarbeit im Studiengang Angewandte Informatik am Institut für Informatik, ZFI-BM-2006-44, ISSN 1612-6793, Zentrum für Informatik, Georg-August-Universität Göttingen, December 2006.

[2] Eclipse. *Eclipse*. The Eclipse Foundation, 2007. Computer Program, http://www.eclipse.org.

[3] E. V. Emden and L. Moonen. Java quality assurance by detecting code smells. In *WCRE '02: Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)*, page 97, Washington, DC, USA, 2002. IEEE Computer Society.

[4] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[5] B. Geppert and F. Rosler. Effects of refactoring legacy protocol implementations: A case study. In *METRICS '04: Proceedings of the 10th International Symposium on Software Metrics*, pages 14–25, Washington, DC, USA, 2004. IEEE Computer Society.

[6] S. Hayashi, M. Saeki, and M. Kurihara. Supporting refactoring activities using histories of program modification. *IEICE - Trans. Inf. Syst.*, E89-D(4):1403–1412, 2006.

[7] Y. Kataoka, T. Imai, H. Andou, and T. Fukaya. A quantitative evaluation of maintainability enhancement by refactoring. In *ICSM '02: Proceedings of the International Conference on Software Maintenance*, page 576, Washington, DC, USA, 2002. IEEE Computer Society.

[8] M. Kersten and G. C. Murphy. Mylar: a degree-of-interest model for IDEs. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 159–168, New York, NY, USA, 2005. ACM.

[9] R. Kolb, D. Muthig, T. Patzke, and K. Yamauchi. A case study in refactoring a legacy component for reuse in a product line. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 369–378, Washington, DC, USA, 2005. IEEE Computer Society.

[10] J. Mankoff, A. K. Dey, G. Hsieh, J. Kientz, S. Lederer, and M. Ames. Heuristic evaluation of ambient displays. In *CHI '03: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 169–176, New York, NY, USA, 2003. ACM.

[11] M. V. Mäntylä. An experiment on subjective evolvability evaluation of object-oriented software: explaining factors and interrater agreement. In *Proceeings of the International Symposium on Empirical Software Engineering*, pages 287–296, November 2005.

[12] E. Murphy-Hill and A. P. Black. Refactoring tools: Fitness for purpose. *IEEE Software*, 25(5), September-October 2008.

[13] 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.

[14] F. Simon, F. Steinbrückner, and C. Lewerentz. Metrics based refactoring. In *CSMR '01: Proceedings of the Fifth European Conference on Software Maintenance and Reengineering*, page 30, Washington, DC, USA, 2001. IEEE Computer Society.

[15] S. Slinger. Code smell detection in Eclipse. Master's thesis, Delft University of Technology, March 2005.

[16] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou. JDeodorant: Identification and removal of type-checking bad smells. In *CSMR*, pages 329–331. IEEE, 2008.

[17] P. Weißgerber and S. Diehl. Are refactorings less error-prone than other changes? In *MSR '06: Proceedings of the 2006 International Workshop on Mining Software Repositories*, pages 112–118, New York, NY, USA, 2006. ACM.

[18] Z. Xing and E. Stroulia. Refactoring practice: How it is and how it should be supported — an Eclipse case study. In *ICSM '06: Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pages 458–468, Washington, DC, USA, 2006. IEEE Computer Society.