2013

Interactive Ambient Visualizations for Soft Advice

Emerson Murphy-Hill

Titus Barik

Andrew P. Black
Portland State University, black@cs.pdx.edu

Interactive Ambient Visualizations for Soft Advice

May 24, 2014

Abstract

Some software packages offer the user *soft advice:* recommendations that are intended to help the user create high quality artifacts, but which may turn out to be *bad* advice. It is left to the user to determine whether the soft advice really will improve quality, and to decide whether or not to adopt it. Visualizations can help the user in making this decision, but we believe that conventional visualizations are less than ideal. In this paper, we describe an interactive ambient visualization to help users identify, understand and interpret soft advice.

Our visualization was developed to help programmers interpret *code smells*, which are indications that a software project may be suffering from design problems. We describe a laboratory experiment with 12 programmers that tests several hypotheses about our visualization. The findings suggest that our tool helps programmers to identify smells more effectively, and to make more informed judgments about the design of the program under development. We then describe an application of our visualization technique in another domain: an English style and grammar advisor. This second application suggests that our technique can be applied to several domains, and also suggests how the technique must be varied to make it domain specific.

Keywords: software, refactoring, code smells, design, soft advice, visualization, ambient, grammar, style

1 Introduction

Liliana is a hypothetical programmer working on the Apache Tomcat project [3]. Recently, she has had difficulty in adding functionality to the JNDIRealm class. This class contains several methods like this one:

```
protected boolean compareCredentials(DirContext context,
    User info, String credentials) throws NamingException {
    ...
    /* sync since super.digest() does this same thing */
    synchronized (this) {
        password = password.substring(5);
        md.reset();
        md.update(credentials.getBytes());
        String digestedPassword =
            new String(Base64.encode(md.digest()));
        validated = password.equals(digestedPassword);
    }
    ...
```

Based on her experience and an inspection of the surrounding class, Liliana has concluded that the context and credentials parameters should be encapsulated into a single object, because these two parameters appear together in the parameter lists of seven different methods. After *refactoring* the code by creating a new class of objects that contain a context and some credentials, then using an object of the new class wherever a context and some credentials appear together, Liliana finds that it is much easier to add functionality to the program, and that her productivity is improved.

How did Liliana recognize that by creating a class now, she would improve her productivity later? The answer is what Fowler calls "code smells" [12]: patterns in programs that make software difficult to build and maintain. Like odors from your kitchen garbage, smells in software suggest (but do not prove conclusively) that something might need attention. In Liliana's case, the smell that she noticed is

Table 1: Some Java Code Smells identified by Fowler [12], by van Emden and Moonen [10] (indicated with *), and by Drozdz and colleagues [9] (indicated with †).

Smell Name	Short Description
DATA CLUMPS	A group of data objects that is duplicated across code.
FEATURE ENVY	A method is more interested in some other class than in its own class.
MESSAGE CHAIN	A series of method calls to "drill down" to a desired object.
SWITCH STATEMENT	A **switch** statement, typically duplicated across code
TYPECAST*	The program makes frequent use of the typecast operation.
INSTANCEOF*	The **instanceof** operator is used to test an object's interface or class.
LONG METHOD	A method is too long to be easily understood.
LARGE CLASS	A class contains too many instance variables or too much code.
PRIMITIVE OBSESSION	The program uses primitive values like **int** and built-in classes like String instead of domain-specific objects like Range and TelephoneNumber.
MAGIC NUMBER†	A literal value is used directly, rather than through a named constant.
COMMENTS	Thickly commented code is often bad code. Refactor the code, and the comments may well become superfluous.
DUPLICATED CODE	The same code structure appears in more than one place.

called DATA CLUMPS: it is produced when the same small group of data objects is used in several different places. DATA CLUMPS can make software more difficult to maintain because if the representation of one of the data objects changes, or the protocol for manipulating those objects changes, then every location in which the group of objects appear must be examined to see if it needs to be modified.

Based on programmers' experience, many smells have been cataloged; Fowler's book lists 22 different smells [12]; other researchers (for example, van Emden and Moonen [10]) have subsequently proposed more smells. Table 1 lists a few interesting smells, including all those mentioned in this paper.

Although smells are intended to help programmers find potential problems with their code, identifying and understanding code smells can be a difficult task

for two reasons.

- First, novice programmers sometimes cannot locate smells as proficiently as experienced programmers, as Mäntylä has demonstrated experimentally [26]. For example, Liliana noticed DATA CLUMPS because of her programming experience and knowledge of the code. A less experienced programmer may not have noticed that there was a problem with the code at all, and may have continued to slowly add functionality to JNDIRealm without understanding the cause of her low productivity.
- Second, even expert programmers can find it burdensome to inspect their code for smells. In Liliana's case, she had to set aside time to manually inspect the code to look for any one of the more than 22 different smells — time that she could have used to add features or fix bigs. Without setting aside time specifically to find code smells, programmers may not notice them.

For these reasons, a class of software tool called a smell detector has been developed to help programmers find code smells and understand their origin. Smell detectors have two parts: a code analysis algorithm, which may be simple or complex, depending on the smell, and a human interface, which presents the results of the analysis to the user.

Smell detectors are an example of a tool that offers what we call *soft advice*. Soft advice is a recommendation produced by a piece of software that is intended to help the user do high quality work, but which may actually be bad advice: the user's expertise is needed to judge whether quality will truly be improved. The recommendations of a smell detector are soft advice because they cannot be blindly applied; an experienced software developer must evaluate the costs and benefits of refactoring away a code smell.

Several other types of tools offer soft advice, outside of the domain of software development. For example, the Firewall Policy Advisor makes recommendations to system administrators for security changes in enterprise networks [2];

the Systems Operation Advisor makes recommendations to aircraft controllers for changes in aircraft fleets [41], and the DB2 advisor makes recommendations to database administrators for database indexes [52]. In each of these soft advice systems, a human must be in the loop to make judgments.

This paper describes a novel smell detector called Stench Blossom that uses an interactive ambient visualization (Section 2). In Section 3 we distill a set of guidelines that capture the important characteristics of Stench Blossom; we believe that these guidelines will be useful to the designers of other smell detectors. We then describe an experiment to evaluate several hypotheses about Stench Blossom and its associated guidelines (Section 4). In Section 5 we describe the application of the idea of interactive ambient visualization to another domain, English writing, and discuss some design implications. Finally, in Section 6, we discuss some potential future improvements for Stench Blossom, and speculate about its broader applicability.

This paper is an extension of a prior conference paper (reference removed to preserve anonymity) where the primary contribution was the design and evaluation of a user-interface for visualizing code smells. However, that work left open an important question: could the visualization techniques that we used for code smells be applied to other domains? This paper answers that question in the affirmative; the major new contribution of this paper appears in Section 5, which describes the application of our visualization technique to a domain outside of software development. (Readers familiar with our original conference publication may safely skip Sections 2 through 4 of this paper, as the content is quite similar.)

2 Stench Blossom: A Novel Smell Detector

In this section, we describe the design of Stench Blossom, drawing from research on refactoring, ambient information display, interface agents, user-interface design, and perceptual attention. We built this tool as a plugin for the Eclipse environment; it can be downloaded at (link removed to preserve anonymity).

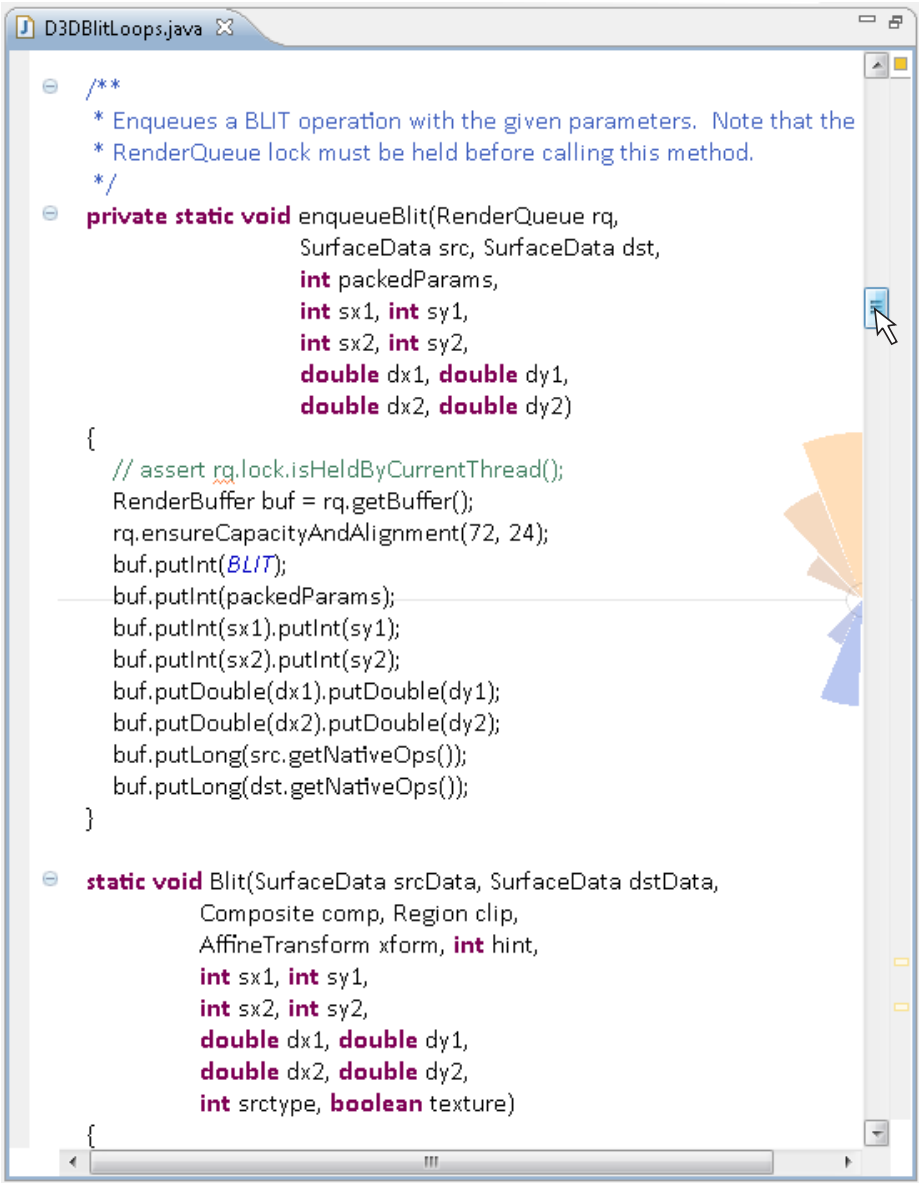

Figure 1: Ambient View in Stench Blossom.

The tool provides the programmer with three different views, which offer progressively more information about the smells in the code being visualized. By default, *Ambient View* is displayed continually while the programmer is coding; it indicates the strength of smells in the programmer's current context, and is illustrated in Figure 1. If the programmer wishes to know more about a particular smell, she mouses over Ambient View to reveal *Active View*, shown in Figure 2; this view names the displayed smells. Finally, if the programmer wants detailed information about a particular smell, she clicks on the smell name in Active View; this reveals *Explanation View*, shown in Figure 3. We discuss each view in turn in the following subsections.

2.1 Ambient View

Ambient View is visible behind the program text whenever the programmer is using the code editor (Figure 1). Likewise, the static analysis engine in Stench Blossom runs silently in the background, so that the information that supports Ambient View is always available. We chose to make the tool constantly available so that it aligns with *floss refactoring*, where programmers frequently switch between refactoring and other kinds of code modification [29]. This design choice is in contrast to that made by smell detectors that must be explicitly invoked to view their results, such as Crocodile [46]; such tools are more appropriate to *root canal refactoring*, where the programmer spends significant, dedicated time refactoring as a software engineering activity separate from other change activities. We feel that aligning our tool with floss refactoring makes it more widely applicable, because floss refactoring is the more common refactoring strategy [30].

The visualization displays smells related to the current programming context. This design decision also derives from our desire to support floss refactoring: our goal is to give the programmer information that will help her carry out her current programming task. A programmer is more likely to be hampered by smells emanating from the code that is the subject of her current programming task than by smells coming from unrelated code, so these are the smells that we choose

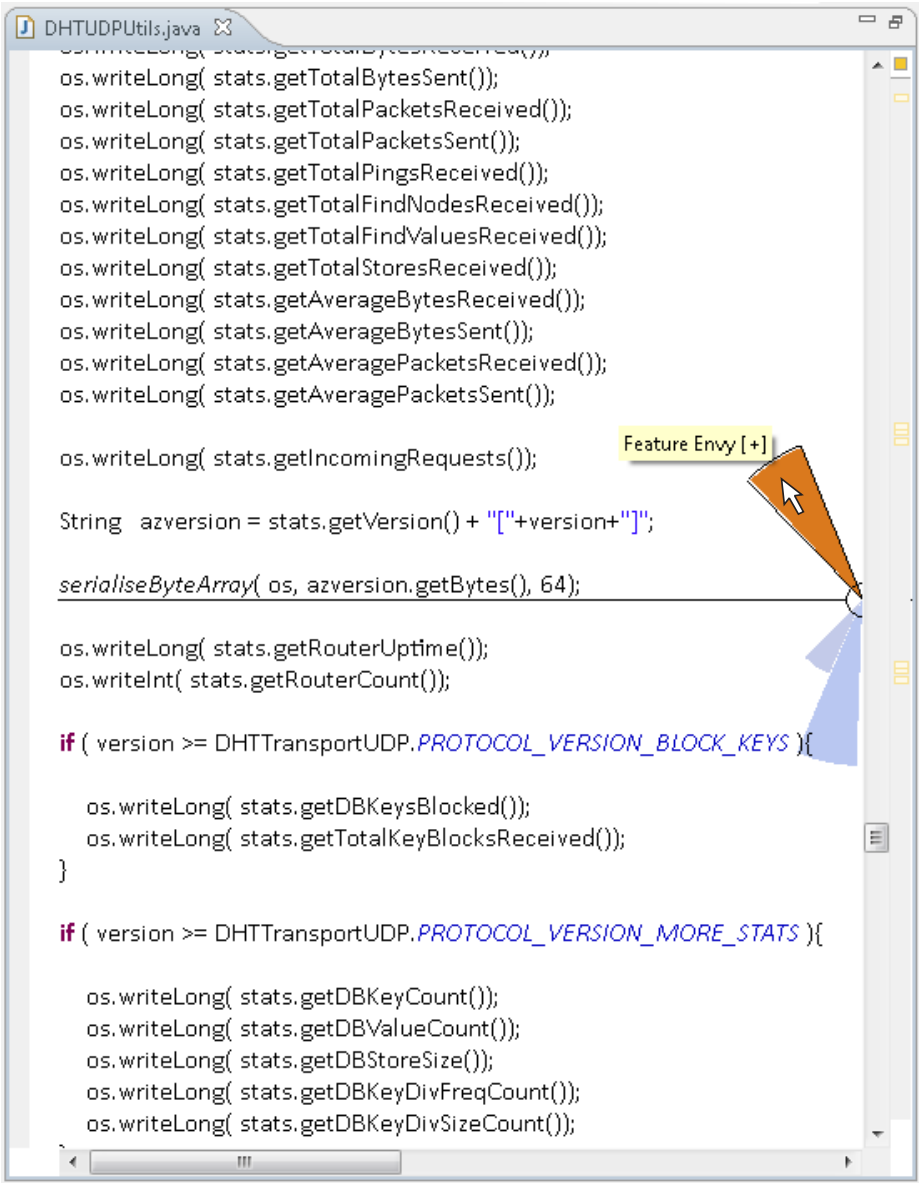

Figure 2: Active View in Stench Blossom.

to display; programmers are also more likely to act to remove smells that come from code that they are going to change anyway. This is in contrast to smell detectors that visualize an entire system, such as jCosmo [10] or CodeCity [54]; these tools are more appropriate for root canal refactoring, where the objective is to find and eliminate the worst system-wide smells to improve overall code quality. Our design choice also aligns with Mankoff and colleagues' recommendation that "the information should be useful and relevant to the users in the intended setting" [25].

The visualization is composed of sectors in a semicircle on the right-hand side of the editor pane. We call these sectors *petals*: each petal corresponds to a smell. We put the petals on the right-hand side of the editor to make the display less distracting: the right-hand side is the part of the pane least likely to contain code. Avoiding distracting developers from their current task was proposed as an important property of ambient software visualizations by Parnin and Görg [38]. We keep the visualization simple, to avoid "information overload" [24]; we do not, for example, display the names of the smells. We chose a fixed, radial pattern for the petals over a more conventional histogram because it may allow users to associate a particular direction with a particular smell, similar to the way in which, after repeated use, users of pie menus can associate items in the menu with a particular direction [6]. For example, a programmer may learn that FEATURE ENVY always appears in the $\nwarrow$ direction. While this circular design limits the number of petals than can be viewed at any one time, we have informally verified that this design scales to at least 20 petals while maintaining readability. Because of the multiple petals and the positions of those petals, our visualization has a *high information capacity* according to Pousman and Stasko's taxonomy of ambient visualizations [40]. At the same time, it uses a *low representational fidelity* [40], because the Ambient View is so loosely coupled to the code, and because the notion of smells is metaphorical. A similar circular visualization using a floral design was *PeopleGarden*, a visualization where a flower represents a person and a petal represents a post to a message board [56]. The Scope tool, which uses

a circular, radar-like visualization to keep the user aware of a variety of system notifications [53], is also similar to our tool.

The length of the petal represents the strength of the smell, where a length of zero represents an absence of the smell, up to the full radius of the semicircle, which represents a very strong smell. For example, in Figure 1, the petal in the ↑ direction (DATA CLUMPS) shows a strong smell, whereas the next petal to its left (FEATURE ENVY) shows a weaker smell. This is in contrast with smell visualizations that use a threshold, such as TRex [32] and CodeNose [47], which don't report smells at all if their metrics fall below a threshold. We made this design decision because we suspect that code smells are highly subjective; if we had chosen a threshold, it would probably differ from the programmer's preferred threshold, with the consequence that the tool will either miss smells that the programmer might want to see (false-negatives), or over-emphasize smells that the programmer would rather ignore (false-positives). Such false-negatives and false-positives may erode programmers' trust in the tool, making them less likely to use it in the future.

Ambient View is drawn in pastel colors *behind* the code, in a fixed position on the screen. Our intent in designing it this way was to make it a frequent reminder and companion during code browsing and editing. In this sense, our visualization uses *negotiated interruption*, where the user is informed of the availability of information but is not forced to acknowledge it immediately [28]. Robertson and colleagues have shown that programmers who use a debugger with negotiated interruption are more productive when completing debugging tasks than are programmers who use a debugger with immediate interruption [43]. Likewise, we hope that Stench Blossom's use of negotiated interruptions, instead of immediate interruptions, will allow programmers to be more productive while programming. Stench Blossom uses a *somewhat low notification level*, according to Pousman and Stasko's taxonomy of ambient visualizations [40].

The light coloration and simple shape of our visualization is also motivated by feature integration theory, which suggests that people initially search in parallel across their entire field of vision for simple visual features, such as color and

orientation, to quickly and automatically perceive objects [50]. After this initial perception, people expend more focused effort to perceive the object in greater depth. We intend that our visualization supports this initial stage of perception, so that programmers can effortlessly assess smells in their code, at least at a high level.

The position of smells in the visualization was designed to give the programmer information at a glance. Each petal, corresponding to a smell, is placed on the semicircle so that the smell that is most obvious to the unaided eye is shown in the $\downarrow$ direction, and the smell that is least likely to be noticed without the use of a tool is shown in the $\uparrow$ direction. This positioning information is replicated in the colors; the most obvious smell is shown in blue, while the least obvious is in orange, with the smells in between colored a gradient between blue and orange. The colors are not strictly necessary to interpret the visualization; the position of the petals alone imparts the information about obviousness. We chose blue and orange because these colors are value-neutral; programmers reported that an early design, where Stench Blossom used red and green, implied that some smells were "worse" than others.

The purpose of the colors and petal positions is that if the programmer notices that the visualization is orange and top-heavy, the code is exhibiting smells that she is unlikely to be aware of, whereas if the visualization is blue and bottom-heavy, the code is exhibiting smells that she is likely to be aware of already. We ranked smells on this "obviousness continuum" because our intuition was that some smells are less obvious than others. For example, a LARGE CLASS is obvious when the programmer is coding within it, yet FEATURE ENVY is less obvious because the programmer needs to determine where each called method or accessed field resides. Stench Blossom displays the smells from top to bottom in the order listed in Table 1 for the first eight smells listed there. This ordering is based on our own intuition as programmers; other programmers may prefer a different order.

By placing smells on the obviousness continuum, we have visually ranked the utility — the usefulness and the importance — of each smell. Gluck and colleagues

have shown that matching the amount of attention attracted by a notification to the utility of the interruption decreases users' annoyance and increases their perception of benefit [14]. We hope that our visual ranking of smells can similarly decrease annoyance and increase the perception of benefit. At the same time, we have designed the user interface so that it avoids distracting the programmer, because, as Raskin puts it, "Systems should be designed to allow users to concentrate on their jobs" [42].

In sum, the purpose of the visualization in Ambient View is to give a lightweight yet information-rich overview of the code smells present in the current programming context. We designed the visualization to impart this information quickly so that the programmer need only glance at the visualization to decide whether further investigation is warranted. We have also built it to be aesthetically pleasing; Pousman and Stasko's taxonomy of ambient visualizations would give it a "somewhat high aesthetic emphasis" [40].

2.2 Active View

If the programmer chooses to investigate a particular smell, she moves the mouse over the offending petal. This transitions Stench Blossom to Active View, and reveals the name of the offending smell, as shown in Figure 2. If she then wants a full explanation of the cause of the smell, she need only click on the name: this transitions Stench Blossom to Explanation View.

We chose to use progressive disclosure to display smell information for two reasons. First, because some types of smell information (such as the information relating to FEATURE ENVY) are highly complex, representing such complexity in a single visualization may be perceptually unscalable. Second, because we wanted Ambient View to be a simple visualization, it was natural to provide the programmer with a way to view in-depth information on demand. Our choice to use progressive disclosure contrasts with other smell detectors, such as Parnin and colleague's Noseprints tool [39], that display a single visualization of code smells. However, many existing smell detectors, especially ones that underline code that

contains smells [17, 32, 47, 51], do include a basic form of progressive disclosure: they allow the user to mouse-over an underlined piece of code to see the name of a smell that that code is exhibiting. Stench Blossom takes this technique one step further in Explanation View.

2.3 Explanation View

Explanation View is designed to explain a selected smell in detail, because simply reporting uniform metrics about the existence and strength of a smell, as we do in the Ambient View, may not be sufficient information to allow the programmer to decide whether and how to correct the smell. This aligns with Shneiderman's recommendation that user interfaces should provide *constructive guidance*, so that the user can make intelligent choices about the next step [44, p. 58].

Although we designed Stench Blossom to provide detailed information about the selected smell, we chose not to offer suggestions for how to refactor the code. We made this design choice for two reasons. First, as Shneiderman states, "experienced operators strongly desire the sense that they are in charge of the system" [45, p. 62]. Second, in some cases, enumerating all the possible refactorings to deal with a smell may yield an overwhelming number of results. For example, given a LONG METHOD, the extract method refactoring may be applied to almost any combination of contiguous statements in the method; each of these refactorings would "carve up" the method in a different way. Instead, Stench Blossom is intended to give the programmer sufficient information to decide for herself on the best course of action.

Naturally, the details provided in Explanation View vary from one smell to another, but most smells are explained using two components. Both are shown in Figure 3, which illustrates the smell FEATURE ENVY.

The first component, initially displayed at the top-right but movable by the user, is the summary pane: it summarizes the data collected by the smell analyzer. In the example, the summary pane shows that the current method uses only a single method (serialiseByteArray) from its own class, but a long list of methods from

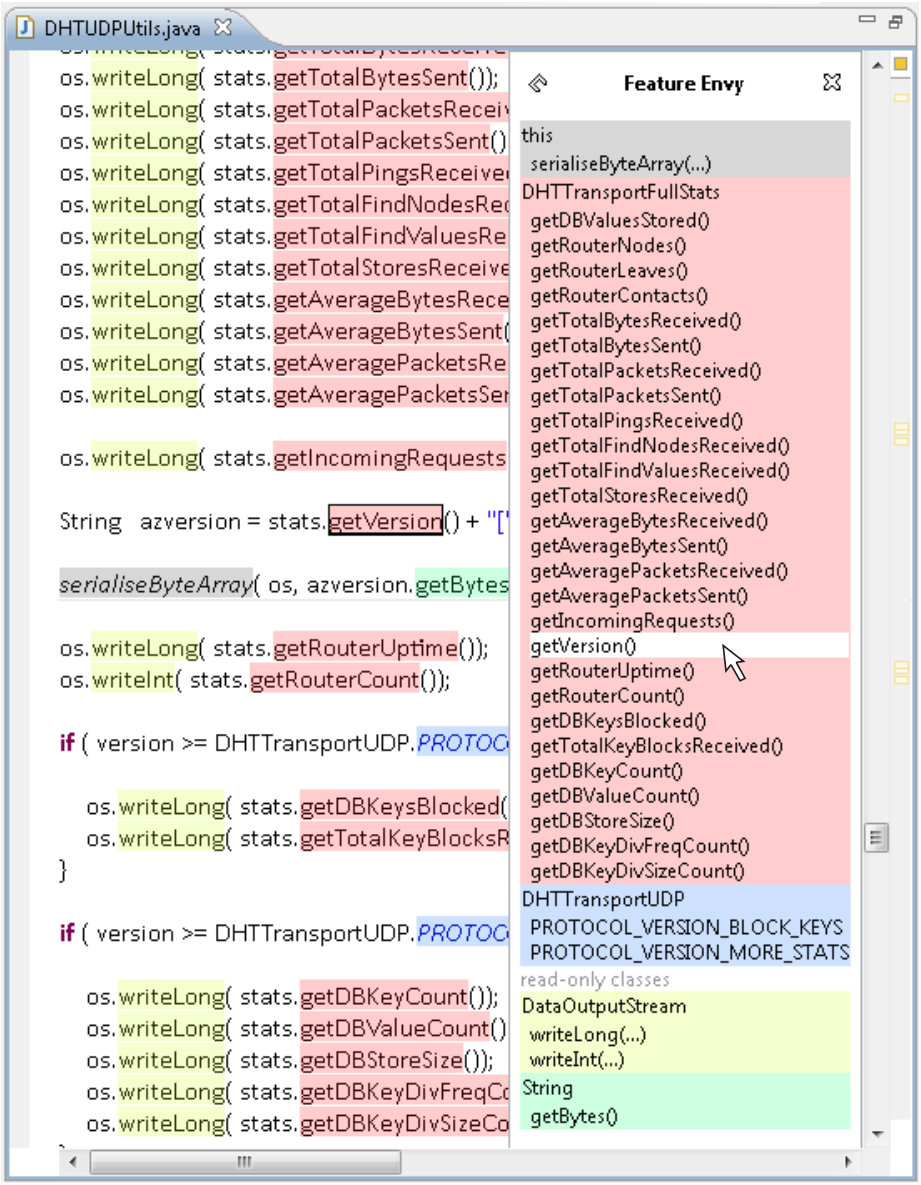

Figure 3: Explanation View for the smell FEATURE ENVY.

the class DHTTransportFullStats.

The second component takes the form of annotations on the code in the editor. These show the origin of the smell. In Figure 3, the programmer has moused-over the name of the getVersion method in the summary pane: the place in the code where this method is used is boxed. References to methods and variables of external classes are highlighted; colors are used to distinguish references to one class from references to another. For example, in Figure 3, all references to methods in DHTTransportImpl are colored pink. Color assignment here is arbitrary, although we attempt to make each color as different as possible from the other colors. We also intend that the programmer can use the overall extent of the colorization to estimate the extent of the smell in the code.

2.4 Implementation

Stench Blossom serves as the common output for a number of individual smell analyzers. Each analyzer computes a scalar metric within a known range, which is used to determine the length of the corresponding petal in the Ambient View. Some of these metrics are quite complicated; the metric for FEATURE ENVY, for example, depends on the number of classes referenced, the number of references to each class, and the number of internal references.

Because of this complexity, care was needed to avoid having the analysis slow-down the response of the system to editing activity, which is after all the primary task. It proved adequate to have smell detection run in a background thread and to cache smell results for unchanged parts of the program. It may eventually prove necessary to rely on heuristics for some analyses in Ambient View, and to commit to a full analysis only if the programmer moves to Active View, and thence to Explanation View.

Showing smells related only to the current programming context — motivated by our desire to support floss refactoring — has the added benefit that it requires more modest analysis than system-wide smell visualizations. This is the key to the scalability of the implementation: even if the program being edited is large, only

a small part of it — the "current context" — is being worked on at any given time. At present, the current context is defined as the method in which the user's cursor lies. If the cursor is not in the code on the screen, then the tool shows a metaphorical trip wire in the middle of the editor; the method on the trip wire defines the current context. In the future, we may consider other definitions of context, such as Mylyn's task contexts [20], or Parnin and Görg's usage contexts [37].

3 Guidelines

Based on our experience designing and building Stench Blossom, we have derived a number of characteristics that we believe may be useful in any smell detector for floss refactoring. In Table 2, we capture these characteristics as a set of user-interface guidelines. The guidelines are stated in a programmer-centric way; these same statements were used in our empirical evaluation. We believe that enumerating these guidelines is important because it captures the characteristics of Stench Blossom in a reusable form; this should help future tool designers to pick and choose which characteristics they want for their smell detector, without necessarily using an interactive ambient visualization. For example, a tool designer who wants to underline smells in an editor could implement *Partiality* by changing the underline color or intensity based on the obviousness of the smell.

Guideline	Rationale
Restraint. The tool should not overwhelm me with the smells that it detects.	Sometimes smells emanate from many pieces of code and sometimes one piece of code emits many smells. For example, the compareCredentials method from Section 1 gives off at least five code smells: DATA CLUMPS, PRIMITIVE OBSESSION, LONG METHOD, COMMENTS, and MAGIC NUMBER. Thus, a smell detector should not display smell information in such a way that a proliferation of code smells overloads the programmer.

Relationality. When showing me details about code smells, the tool should show me the relationships between affected program elements.	Some smells emanate not from a single point in the code, but from the *relationship* between several non-contiguous pieces of code. For instance, a method exhibits FEATURE ENVY not because of a single reference to a parameter, but because of a large number of references to parameters of foreign types and a small number of references to the fields and methods of **this** and other objects of the method's own class. Thus, a smell detector should display smell information relationally when the smell is caused by the relationship between code fragments.
Partiality. The tool should emphasize smells that are difficult to see with the naked eye.	Programmers may find that there is more value in having a tool tell them about certain smells and less value in being told about other smells. This is because some smells, such as LONG METHOD, are visible at a glance, while others, such as FEATURE ENVY, require careful analysis [26]. Thus, a smell detector should emphasize those smells that are difficult to recognize without a tool.
Non-distracting. The tool should not distract me.	It is important that a smell detector not encourage a programmer to refactor excessively, because best practice dictates that programmers only refactor when it helps achieve another goal [12].
Estimability. The tool should help me estimate the extent of a smell in the code.	Smells such as DUPLICATED CODE may be spread throughout a whole class whereas others may be localized in only one place. The extent of such spread can help the programmer determine whether or not a smell should be refactored away, and how much effort and reward such a refactoring would entail.

Availability. The tool should make smell information available to me at all times.	The most popular tactic for refactoring occurs when a programmer interleaves frequent refactoring with other kinds of program modification — floss refactoring [30]. Because analyzing smells is part of this interleaving, a smell detector that supports this tactic must help programmers to find smells quickly, without forcing them to go through a long process to see if the tool has found any smells. Thus, a smell detector should make smell information available as soon as possible, with little or no effort on the part of the programmer.
Unobtrusiveness. The tool should not block me from my other work while it analyzes or finds smells.	The activities of coding and finding smells for refactoring are often tightly interleaved [12, 30], yet at the same time, automatic code analysis may be time consuming, so much so that waiting for the analysis to complete may disrupt this interleaving. Thus, a smell detector should should not stop the programmer from programming while it gathers, analyzes, and displays information about smells.
Context-Sensitivity. The tool should tell me first and foremost about smells related to the code I'm working on.	Best practice dictates that refactoring only be done when it helps to accomplish an immediate programming goal [12]; fixing a smell on code that is unrelated to the current programming task is a distraction from that task. So fixing smells in a context-*insensitive* manner may be an inefficient way of using resources, or may even be counter-productive. Thus, a smell detector should point out smells relevant to the current programming context.

Lucidity. In addition to finding smells for me, the tool should tell me why smells exist.	Smells can be complex and difficult to understand, because they may be subtle or flagrant, widespread or localized, or anywhere in between. A smell detector that communicates these properties may help give the programmer confidence in the detector's analysis. Thus, a smell detector should go further than simply telling the programmer that a smell exists; it should help the programmer find the sources of the problem by explaining *why* the smell exists.

Table 2: Our guidelines and the rationale behind them.

4 Experiment

We conducted an experiment to test several hypotheses about Stench Blossom. In the experiment, we asked programmers to identify smells in code and make refactoring judgments based on smells, with and without Stench Blossom. To facilitate replication of this experiment, the experimenter's notebook can be found in the Appendix. Other materials, including the codesets and results database, can be found at (link removed to preserve anonymity).

In designing the experiment, we chose to compare Stench Blossom against no tool, rather than comparing it against some existing tool. While it would be useful to compare different smell visualizations, no such comparison could in practice be fair. Existing smell detectors differ from ours in that they work for other languages or for considerably fewer smells, and thus the results of such a comparison would necessarily conflate the effects of those differences with differences between visualizations. For instance, van Emden and Moonen's tool [10] implements only two smells (INSTANCEOF and TYPECAST); conducting a comparative experiment against just these two smells would produce quite limited results.

Class	Job Title	Experience	Tool 1^{st}	Code	Smells Found	
					No Tool	Tool
No	Software Analyst	7 years	Yes	AB	11	18
No	Software Developer	30 years	No	AB	9	14
No	Advisory Software Eng.	10 years	Yes	BA	12	22
No	Senior Software Eng.	18 years	No	BA	20	35
Yes	IT Support	9 years	Yes	AB	17	27
Yes	Graduate Teaching Asst.	6 years	No	BA	11	23
Yes	Software Eng.	4 years	Yes	BA	22	20
Yes	Technologist	19 years	No	AB	11	22
No	Software Eng.	15 years	Yes	BA	14	18
Yes	Hardware Validation Intern	3 years	Yes	AB	8	13
Yes	Systems Eng.	5 years	No	BA	9	17
No	Contract Programmer	10 years	No	AB	10	22

Table 3: Participant demographics and results summary.

4.1 Subjects

We recruited a total of 12 subjects: 6 commercial Java developers and 6 students from a graduate class on relational database management systems. Subjects were recruited using an email message that stated that participants needed to be at least moderately familiar with Java, and unfamiliar with Stench Blossom.

Subjects from the class were asked to volunteer to participate in exchange for extra credit on one programming assignment. Professional subjects were drawn from a pool of local professional programmers who had volunteered previously at Java and Eclipse user group meetings. Professional subjects were not compensated.

Based on self-reporting in a pre-experiment questionnaire, it appeared that subjects arrived with the requisite amount of programming experience, and a varied level of experience with refactoring and smells. All subjects had previously used integrated development environments (9 of 12 were Eclipse users), and were at least moderately familiar with Java. All professional subjects had some knowledge of refactoring, while four out of six student subjects did. Four out of six professional subjects had some knowledge of smells, while none of the student

subjects did. Professional subjects reported a median of 12.5 years of programming experience, while student subjects reported 5.5 years.

Table 3 summarizes the demographics of the subjects, where each row represents a subject. The **Class** column indicates whether the subject was recruited from the graduate class. The **Job Title** column indicates each participant's job title. The **Experience** column indicates how many years of programming experience each subject had. The remainder of the columns in the table will be explained shortly.

4.2 Methodology

We conducted the experiment using a laptop (1.7 GHz, 2GB RAM, 15.4 inch-wide screen of 1280×800 pixels) with an external mouse. Each experiment was conducted one-on-one, with the first author as experiment administrator.

Subjects were divided into four groups to mitigate learning effects via counterbalancing. Half of the subjects performed tasks without the aid of Stench Blossom first, then with the aid of Stench Blossom, while the other half used Stench Blossom first, then performed the task without it. In Table 3, the **Tool 1**st column indicates whether subjects used the tool first. Within these two groups, half of the subjects worked over codeset A first, then B second, and half over codeset B first, then A second. In Table 3, the **Code** column indicates this order for each subject. We chose codesets A and B to contain an approximately equal variety of smells. Each codeset contained 4 classes selected from the Vuze [48] and core Java libraries [36].

We designed the experiment to last approximately one hour. The experiment started with a training phase, then had three parts in which we tested four hypotheses, as described below.

4.2.1 Training

Subjects were given eight $3'' \times 5''$ cards, each containing a smell name and description on the front, and an example on the back. The eight smells on the cards were the first eight smells listed in Table 1. Subjects were given a few minutes to

read these cards, and were told that they would later be asked to find smells as well as explore details of some smells.

4.2.2 Task 1: Identifying Smells in Code

Subjects were asked to skim four Java files, top to bottom, and mention any smells that they noticed. For two of the files, subjects looked for the smells manually, and for the other two they used Stench Blossom. Before using Stench Blossom, the administrator gave each subject a demonstration and read aloud a short description of the Ambient View visualization.

The subject then began the task, and the administrator recorded which of the 8 smells the subject noticed and said aloud, with and without Stench Blossom. When using Stench Blossom, the subjects could simply repeat what the visualization was telling them, but in practice, most subjects appeared to cross-validate what the tool said by briefly looking over the code. This allowed us to test our first hypothesis:

Hypothesis 1 *Programmers identify more smells using the tool than not using the tool.* If the number of smells that subjects reported when using the tool significantly exceeds the number of smells when subjects were not using the tool, then the hypothesis is confirmed.

Note that, while this hypothesis may seem obviously true, little evidence exists in the literature to confirm it. The only experiment that we know of that has tested this hypothesis was performed by Parnin and colleagues [39], where one of the authors found more smells using a tool in a small software project than did five other code readers without a tool. Thus, our confirmation of this hypothesis serves to confirm Parnin and colleagues' result, for an audience beyond the people who designed the tool: smell detectors can be effective in finding smells.

We also asked subjects to evaluate, and say aloud, whether they agreed with the tool's quantification of the smell. This allowed us to test another hypothesis:

Hypothesis 2 *Code smells are subjective.* If subjects expressed disagreement with each others' estimation of smells and with the tool's quantification of the

smell, then the hypothesis is confirmed.

Previous evidence for this hypothesis has been provided by Mäntylä and colleagues, who asked 12 developers from the same company to identify smells in their own closed-source software and compared that evaluation to a smell detector's findings for three smells [27]. They found that the findings of the developers and the findings of the tools did not correlate, confirming this hypotheses. Our study thus attempts to qualitatively replicate their findings in the context of more smells, a wider variety of programmers, and for open-source software.

4.2.3 Task 2: Making Refactoring Judgments

Next, subjects made refactoring judgments about code. When the subject used Stench Blossom, the administrator gave the subject a demonstration of the tool and read aloud a description of how the Explanation View displays FEATURE ENVY. The subject was then told the task was to "use the tool to help you make some judgments about the code: how widespread the FEATURE ENVY is, how likely you are to remove it, and how you might do it." The subject performed this task in four different methods: two methods with Stench Blossom, and two methods without. We recorded these judgments during the experiment. A similar task description was used when the subject did not use the tool.

We used this task to evaluate the following hypothesis:

Hypothesis 3 *Programmers make more confident and informed refactoring judgments when using the tool than when not using the tool.* In the questionnaire (described in the next section), we asked subjects whether they felt that the tool helped them to make more confident and informed judgments. These two questions about being confident and informed allowed yes-or-no responses with optional comments. If the number of subjects who reported being more confident and informed about their judgments exceeded the number who did not, then the hypothesis is confirmed.

4.2.4 Questionnaire

Finally, subjects were asked about their experiences using Stench Blossom, and about their opinion of smell detectors in general.

In the questionnaire, we also asked subjects to rate whether the nine usability guidelines (described in Table 2) were important. Similar to a heuristic evaluation, where people evaluate a user interface according to a set of guidelines [34], our ratings instead tried to have subjects evaluate the guidelines themselves. By phrasing the guidelines in a programmer-centric way we hoped that subjects could judge whether each guideline was important to them personally. Because the two tasks allowed the subject to explore the breadth and depth of our tool (several smells and all three views), we feel that the subjects were qualified to make an informed judgment about our tool, and of the guidelines that it follows.

Additionally, the questionnaire asked subjects to rate two other guidelines that a smell detector might exhibit but that we did *not* postulate in Section 3, and which Stench Blossom does not follow.

- *Decidability*, "The tool should help me decide whether to remove a smell from the code", similar to Shneiderman's recommendation for constructive guidance [44, p. 58].
- *Consistency* "The tool should have a user interface consistent with the rest of the environment," derived from Nielsen's "consistency and standards" heuristic [33].

We included these two guidelines because we postulate that they are *not* important to smell detectors. Thus they provide a baseline against which to test the guidelines that we do postulate to be important. This leads to our fourth hypothesis:

Hypothesis 4 *The guidelines represent desirable design considerations for smell detectors.* If subjects rank the guidelines that we believe are important to smell detectors significantly higher than the guidelines that we believe are not important, then the hypothesis is confirmed.

4.3 Results

The experiment confirmed **Hypothesis 1**, that programmers identify more smells using the Stench Blossom tool than not using it. In Table 3, the **Smells Found** columns indicate the number of smells identified with and without Stench Blossom. The median number of smells found without the assistance of Stench Blossom was 11, while the median number of smells found with the assistance of Stench Blossom was 21. The difference between smells found with Stench Blossom and those found without is statistically significant ($p = .003$, $df = 11$, $z = 2.98$, using a Wilcoxon matched-pairs signed-ranks test, where $\alpha = .05$). This aligned with subjects' opinions: all indicated that it was difficult to look for all 8 smells at once without the assistance of the tool. All subjects indicated that the smell detector found information that they would not have found as quickly. Eight of the twelve indicated that the detector found information that they would not have found at all.

When subjects did not use Stench Blossom, they sometimes found the task of recognizing smells difficult, suggesting that one factor that made Stench Blossom effective was that it served as memory aid. When the administrator asked subjects to look for the 8 smells in the code, subjects reported that they found it difficult to keep them all in mind at once. Overall, 4 subjects "somewhat agreed" and 8 "strongly agreed" that "it was difficult to look for all 8 smells at the same time." While looking for smells, a subject remarked "I realize [that] I forgot about the LONG METHOD one" and "TYPECAST: I'd totally forgotten," even though this subject had reviewed the smells less than 10 minutes prior and was among the 3 subjects who rated themselves most knowledgeable about code smells. Likewise, even when readily apparent by inspection, some smells were overlooked by subjects. For example, after overlooking a **switch** statement several times, one subject commented "I can't believe I didn't see it."

The experiment provided evidence to support **Hypothesis 2**, that smells are subjective. For example, several subjects had different definitions of what "too big" means for LONG METHOD and LARGE CLASS. Several subjects agreed with

Stench Blossom — that counting the number of characters is useful for gauging how long something is — although some commented that the tool should not have included comments when gauging size. Other subjects stated that counting statements or expressions in the abstract syntax tree is the only useful metric for length. One subject noted that "if it fits on the page, it's reasonable." There was some indication, beyond LONG METHOD and LARGE CLASS, that other smells were subjective as well. For instance, one subject saw some instances of DATA CLUMPS as not a problem because the developers who wrote the code had little choice. Likewise, subjects made comments indicating that smells were not binary, but encompassed a range of severity; for instance, smells were "borderline," "obvious," or "relative" to the surrounding code.

The experiment confirmed **Hypothesis 3**, that subjects make more confident and informed refactoring judgments when using the tool than when not using the tool. 10 out of 12 subjects said that the tool improved their confidence in refactoring judgments, and 11 out of 12 said that the tool helped them to make more informed judgments.

A feature that appeared to help subjects make refactoring judgments was Stench Blossom's ability to perform and express precise program analysis. Without the tool, several subjects inaccurately analyzed source code for FEATURE ENVY, which led to poorly informed refactoring judgments. The inaccuracy of the subjects' analyses appeared to stem from their use of faulty heuristics. For example, one subject explicitly declared a heuristic that if the method being inspected "is static ... [then] we're not referencing ... this class." This heuristic, used by several developers, is faulty because static methods can access static fields. Using this heuristic will cause subjects to conclude that there is more FEATURE ENVY than actually exists, potentially resulting in unnecessary refactoring. Because Stench Blossom performed accurate program analysis, subjects did not need to rely on faulty heuristics, and thus could make refactoring judgments that were confident and informed.

The experiment confirmed **Hypothesis 4**, that the guidelines represent desir-

How important is the characteristic to any smell detection tool?	Not Important	Somewhat Important	Important	Very Important	Essential
Unobtrusiveness	0	0	0	1	11
Context-Sensitivity	0	1	1	3	7
Restraint	0	1	1	3	7
Partiality	0	1	0	6	5
Estimability	0	0	3	3	6
Non-distracting	0	1	1	5	5
Relationality	1	1	3	4	3
Availability	1	2	2	4	3
Consistency	1	2	2	5	2
Lucidity	3	0	3	3	3
Decidability	3	2	4	2	1

Table 4: Results of post-experiment guideline questionnaire.

able design considerations for smell detectors. Overall, subjects rated our guidelines as important to the design of smell detectors. Table 4 lists how subjects rated each guideline that we postulated in Table 2. In the left column, the guideline name is listed (the subject had read the description of the guideline, but not the name of the guideline). The right columns list how the many subjects rated each guideline at what level of importance to the design of smell detectors. For example, 1 subject marked Unobtrusiveness as "Very Important" while 11 marked it as "Essential." The aggregates of all responses are displayed; the darker the table cell, the more participants marked that response. In the table, guidelines are ordered by mean guideline scores. Guidelines that were not included in the originally postulated list of 9 guidelines are *italicized* in Table 4. Subjects tended to rank the

	Consistency	*Decidability*
Unobtrusiveness	<.001*	<.001*
Context-Sensitivity	.042*	.003*
Restraint	.042*	.003*
Partiality	.065	.003*
Estimability	.086	.004*
Non-Distracting	.107	.006*
Relationality	.742	.086
Availability	.834	.125
Lucidity	.858	.286

Table 5: Differences between postulated guidelines (at left) and guidelines that we did not postulate (top), compared using pair-wise p-values calculated using a Wilcoxon-rank-sum test. Statistically significant differences denoted with a *, at a $\alpha = .05$ level.

postulated guidelines, as a whole, significantly higher than the guidelines that we did not postulate ($p < .001$, $df = 130$, $z = 3.69$, using a Wilcoxon rank-sum test), suggesting that programmers believe that our guidelines are generally important to usable smell detectors. However, individually, only the Unobtrusiveness, Context-Sensitivity, and Restraint guidelines were ranked significantly higher than both guidelines that we did not postulate (Table 5).

A minority of subjects appeared to believe that some guidelines are not at all important. For example, the postulated guideline that was judged the least important, lucidity, was judged as "not important" by 3 subjects. Interestingly, these 3 subjects were all volunteers from the classroom, and were the second, third, and fourth least experienced programmers among the 12 subjects. Our interpretation is that, perhaps, less experienced programmers do not value a tool that explains its reasoning because they believe that needing such an explanation is a sign of poor programming skills.

4.4 Limitations

There are several limitations in the design of our experiment. We restricted subjects to discussing only 8 smells, when Fowler lists 22 code smells [12], and those 8 are not necessarily a representative sample. Likewise, we only focused on one

smell in the Explanation View — FEATURE ENVY — so subjects' refactoring judgments may be different for other kinds of smells. For the most part, subjects were unfamiliar with the source code; the results may be different for code with which they are familiar. A further limitation is that the source code that we selected may not be representative of all source code in the wild. Further studies are needed to validate our results for different smells, with code familiar to subjects, and with a wider variety of code bases.

5 Visualizations for Grammar Smells

In this section, we describe how we repurposed Stench Blossom as a style and grammar advisor. Our goal was to demonstrate the generalizability of interactive ambient visualizations by applying them to a domain outside software engineering. Revision in writing is an appropriate avenue for such a tool because a revision task can occur at any time in the subprocesses of writing, as opposed to being linearly performed in a single postwriting stage of the composing process [11].

5.1 Implementation

To closely parallel the implementation of visualizing code smells, the style and grammar advisor also uses Eclipse as its underlying editing environment. Specifically, we use TeXlipse [19], a plugin for editing LaTeX documents, to provide a convenient platform for retargeting Stench Blossom to English writing, rather than programming. This implementation decision also allows us to reuse much of the existing visualization framework.

The retargeting of Stench Blossom as a style and grammar advisor is greatly facilitated by the use of the open source style and grammar checking library JLanguageTool [31]. Designed by Daniel Naber, this library offers the ability to grammar check plain-text documents in multiple languages and includes an ample number of rule-based grammar patterns.

The fact that JLanguageTool processes plain-text presents an interesting pars-

ing challenge, since LaTeX is a markup language and LaTeX documents contain markup as well as text: special characters, variables, command and environment definitions and applications, and comments. Precautions must be taken before passing such a document to the JLanguageTool grammar checker by preprocessing the input LaTeX text. Specifically, LaTeX tags are stripped of programming commands before grammar-checking the document; these commands are then added back after checking. This requires that we adjust the character positions to compensate for the grammar checker's changes. While this eliminates many issues, such as formatting commands like bold or underline, our transformation does not actually compile or execute the LaTeX code. Consequently, if the LaTeX document contains programming statements that add or remove content from the document itself, these statements cannot be detected, causing incorrect input text to be sent to JLanguageTool. This issue is not unique to our implementation, but is inherent to the fact that LaTeX is a Turing-complete programming language. The spelling engine of TeXlipse, for example, is similarly unable to spell-check text in macros.

JLanguageTool first performs part-of-speech tagging; the core rule-checking engine then uses part-of-speech information and regular expressions to encode grammar rules. This type of grammar error detection and recommendation mechanism is thus roughly analogous to a generalized search and replace mechanism. This results in some limitations in the tool's ability to recognize and encode complex grammar issues, such as incorrect parallel structures.

Gamon summarizes the power of this form of recognition by identifying two extremes [13]. The first is that of preposition and article errors, which require large amounts of contextual information to arrive at a correct identification. The second is that of overregularized verb inflection, and is detectable without any contextual information. As an example of the second extreme, the word `goed` is incorrect, even without knowing the context; the word should be changed to `went`. Regular-expression based tools like JLanguageTool implement more checks in the latter category, and fewer in the former. While the library offers the ability to write advanced contextual rules in Java, few such rules are provided in the JLanguageTool

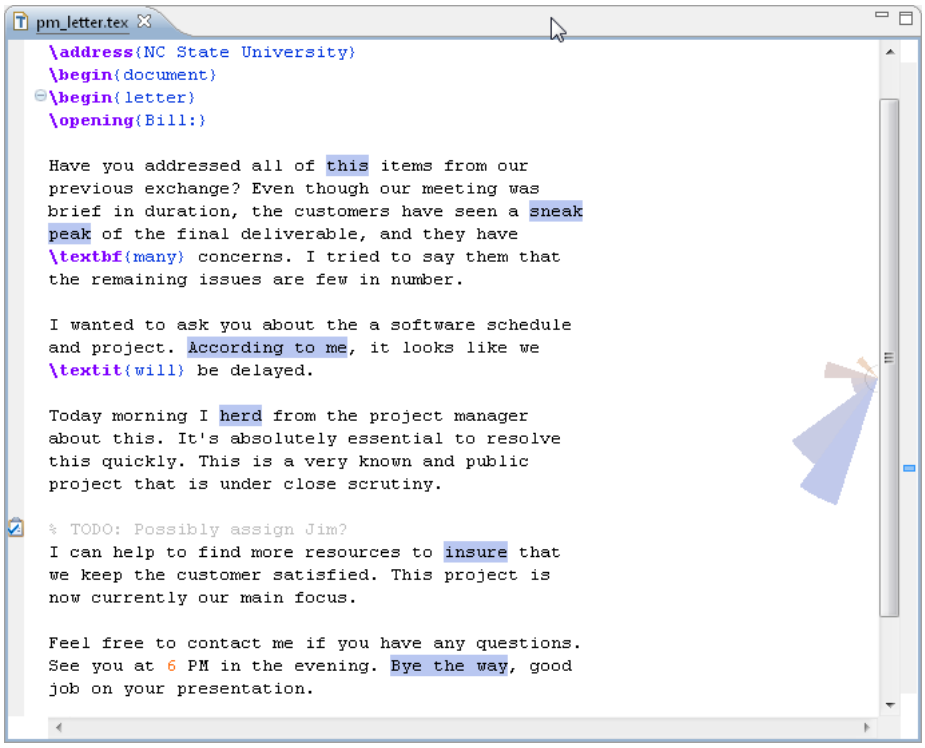

Figure 4: Stench Blossom displaying grammar issues for the categories General Grammar and Possible Typos.

implementation. Implementing our visualization on top of these rules still provides useful information, although more advanced visualizations could be developed if more sophisticated grammar rules become available.

JLanguageTool places grammar rules into a small number of distinct categories, so it was convenient to map each category to a separate petal. Within each category, the number of rules violated by the text in the visible document region determines the length of the petal. Categories include bad style, commonly confused words, nonstandard phrases, and possible typos; Table 6 shows a complete list of categories, along with representative rules from each.

5.2 Usage

Table 6: Grammar categories within JLanguageTool and example rules for each.

Category	**Description**	**Incorrect Usage**	**Correct Usage**
Bad Style	Starting a sentence with a numeral, starting a sentence with “Hopefully”, and three nouns in a row.	**12** soldiers were killed!	**Twelve** soldiers were killed!
Capitalization	Corrects case of incorrectly lowercased and uppercased words.	Who do you think **i** am?	Who do you think **I** am?
Collocations	Suggests collocations — associated words that often appear together.	I’m going home **today evening**.	I’m going home **this evening**.
Commonly Confused Words	Words that sound alike, but that have different meanings.	Please **except** my apologies.	Please **accept** my apologies.
General Grammar	General grammar issues involving agreement and incorrect sentence structure.	The dogs **barks** loudly.	The dogs **bark** loudly.

Table 6: Grammar categories within JLanguageTool and example rules for each.

Category	**Description**	**Incorrect Usage**	**Correct Usage**
Miscellaneous	Miscellaneous grammar issues, including repetition of words, double negations, and other types of duplicated words and phrases.	**The a** thing is this.	**The** thing is this.
Nonstandard Phrases	Phrases that are non-standard for native English speakers, but commonly said by non-native speakers.	No, **in the moment** not.	No, **currently** not.
Possible Typos	Possible typos in words determined by context, which would not be detected using spell checking alone.	Do not duplicate or redistribute **in any from**.	Do not duplicate or redistribute **in any form**.
Redundant Phrases	Redundant expressions — the removal of one expression would be more concise without resulting in loss of clarity.	Their voices **blend together** in lovely harmony.	Their voices **blend** in lovely harmony.
Slang	Slang word or phrase is used in a formal context. Currently, JLanguageTool contains only a single rule in this category.	Did you receive **an invite**?	Did you receive **an invitation**?

Figure 4 shows a typical editing session within the TeXlipse environment. Hovering over the petal reveals a tooltip with the name of the category of that petal. The default behavior is that clicking on the name of the category toggles the Explanation View, which highlights the grammar issues within the visible document region. The user can opt to display the Explanation View for as many categories as she desires.

Obviousness for grammar issues is presented in a manner similar to code smells, with the least obvious grammatical errors presented in the $\uparrow$ direction, and the most obvious errors presented in the $\downarrow$ direction. While code smells used a variety of different metrics for determining the size of a petal, the grammar visualization uses only the number of visible issues in the category to determine the size. We discuss the difficulty of designing appropriate metrics for obviousness and severity in the next two subsections.

5.3 Determining Obviousness

For code smells, we determined the obviousness of each smell using our experience as software developers. In doing so, we made the assumption that obviousness is an objective quality. For the grammar advisor, that is not so, and thus the ordering of categories by obviousness raises some new difficulties.

Evidence for the subjective nature of obviousness for errors in natural language comes both from our everyday experience and from the linguistics literature. Han, for instance, states that one of the most complex problems faced by a non-native speaker of English is the correct use of articles (a, an, and the), and that using articles correctly is particularly difficult for speakers of Japanese, Chinese, and other languages that do not have articles [16]. It seems reasonable to conclude that a native English speaker might find issues of article agreement to be obvious, whereas a non-native speaker may find them quite non-obvious, indeed immensely difficult to detect without the aid of a tool.

A possible response might be to have the visualization use different scales of obviousness for native and non-native writers, as determined by analyzing the text.

This too is fraught with difficulties; it turns out that native and near-native speakers may not exhibit significant differences in competence. For example, Coppieters performed extensive interviews of French speakers and found that native and near-native speakers of French have strikingly different intuitions on French sentences, in spite of the fact that the two groups appear to be equivalent at the level of language use and proficiency [7].

One solution is to allow the user to determine the ordering of the categories based on individual preferences. Indeed, this is exactly what we have done; the current obviousness rankings are a reflection of the authors' preferences as native English speakers. Other solutions include asking the user directly ("Are you a native speaker?"), detecting the appropriate obviousness order based on the frequency with which the user makes certain grammatical mistakes, or by providing a training session that scores the user's ability to recognize grammar issues.

5.4 Determining Severity

Obviousness is only one metric in the design space of our visualization. It alone tells us nothing about the severity of the problem. Recall that the severity has a visual effect on the petal: increased severity corresponds to increased petal size. To analyze severity, it seems reasonable to look at which of the individual rules within a category have been violated. But how should we decide which rule violations are trivial, and which serious?

As with obviousness, there seems to be no clearcut metric for severity. A plausible approach to determining the severity of each grammar rule might be to examine the literature on academic grading of writing samples. Unfortunately, there appears to be wide variability in this area. Lee, for example, investigated the grading behavior of Korean and native English-speaking graders, and found that Korean graders were more severe in scoring grammar, sentence structure, and organization, whereas the native English graders were stricter on content and overall scores [22]. Similarly, Derwing and Rossiter conducted a study of native and non-native speakers that identified grammatical errors in sentences and rated them

on gravity and annoyance, two measures of severity. Their findings show differences between the two groups due to error salience and language awareness [8]. They present evidence that non-native speakers were significantly more annoyed by a high frequency of errors than native speakers, regardless of the kind of error, and that non-native speakers considered all errors to be more serious than native speakers.

For the purposes of visualization, we can approach this issue from two opposite directions. The first direction is to consider modifying the weighting of a rule so that the tool's assessment of severity more closely matches the expectations of the user. The second direction is to have a panel of language experts decide on the severity of each error, to use this ranking to fix the weighting of each rule, and to use the tool as means to train the user by informing them of the "actual" severity of their grammar errors, which may or may not align with their perceived idea of the severity. In short, we can either let the user train the visualization, or let the visualization train the user.

Rather than deliberate over the most appropriate solution for classifying severity, our implementation simply uses the count of the number of issues in that category to determine the length of the petal; each rule violation is given equal weight. We use a linear scale with a maximum cap; through trial and error, it appears that a linear calculation for petal size is more appropriate for grammar issues than the logarithmic calculation that we use for code smells. We speculate that this may be because code has more repetitive elements due to its explicit structure and limited vocabulary when compared with the subtleties of natural language. Certainly, the issue of assigning severity to grammar errors is an open problem that merits further investigation.

5.5 Visualizing Redundant Phrases

Most of the possible errors detected by JLanguageTool can be visualized quite adequately using simple highlighting in the Explanation View, as shown in Figure 4. This is because most of the tool's rules simply point out a problem word

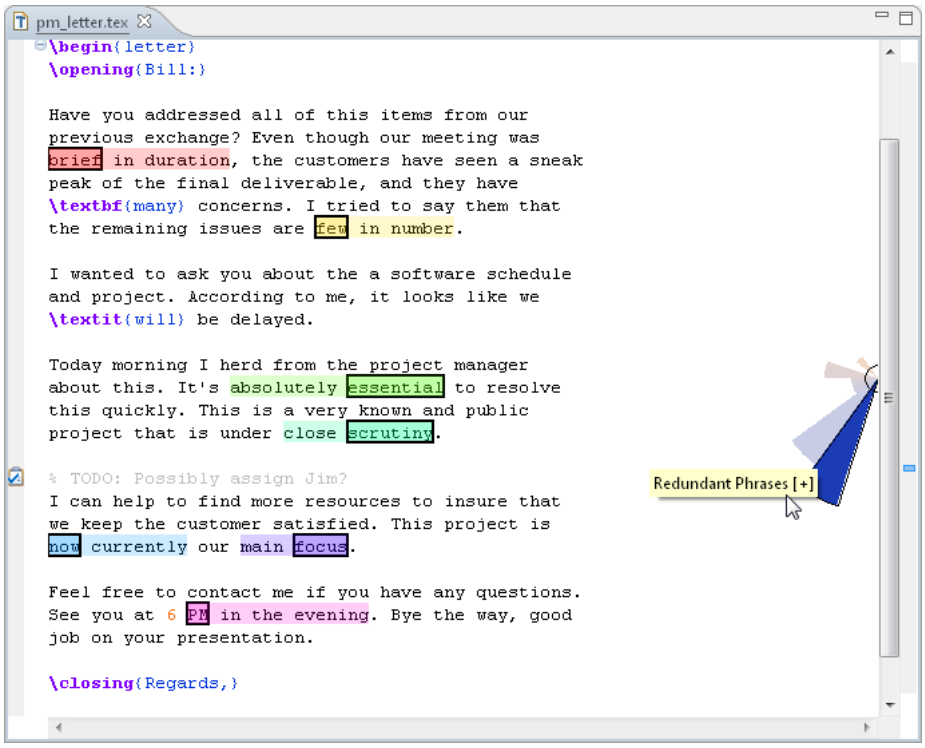

Figure 5: Stench Blossom displaying an alternative visualization for Redundant Phrases.

or phrase. However, a few rule-based patterns lend themselves to more expressive visualizations, among them, *redundant phrases*.

In its simplest form, redundant phrasing, also known as pleonasm, can be described as a pair of expressions in which one of the pair can be removed without changing the overall meaning, such as the phrase “rustic country”. Redundant phrases are interesting in that while they have recommended replacements, small changes to the context surrounding the sentence can often allow the writer to choose either member of the redundant pair. For instance, in the phrase “ATM machine”, one could either remove the word machine, or replace the abbreviation “ATM” by “automated teller”. The phrase “PM in the evening” could either be replaced by “PM” or “in the evening”. More complicated redundant phrases require more context. The redundant phrase “foreign import” could be replaced by either

"foreign" or "import". Here, the selection of expression depends on context. If the context is the prefix "is", then the only choice is "foreign". If, on the other hand, the context is "is an", the only possible choice of expression is "import".

Because the user must choose between two resolutions, and because redundant phrases occur as pairs of words or expressions, we provide a more informative visualization: Figure 5 shows the result of activating Explanation View for the Redundant Phrases petal. The entire redundant phrase is highlighted in color, but the expression that the JLanguageTool rule recommends be kept is surrounded by a box.

Our visualization of redundant phrases also demonstrates the ability of Stench Blossom to encode additional information within its visualizations. Instead of simply indicating an error with highlighting, it is possible for the system to provide more information about the error, such as a suggested fix. If necessary, a separate visualization technique can be used for each category of error.

Having described our implementation of a grammar advisor in stench blossom, and examined some of the ways that the grammar domain differs from code smells, we now consider how our visualization technique differs from existing approaches to displaying grammar issues.

5.6 Differences from Grammar Advisors and Writing Activity Visualizations

In this section, we contrast Stench Blossom for grammar issues with existing approaches, such as the wavy underlines used by word processors like Microsoft Office and OpenOffice. At first glance, it may appear that our implementation is similar to these systems in the way it alerts the user to possible errors. However, there are key differences.

While the wavy underlining technique shows that a grammar error exists, it does so for all errors on screen simultaneously. Given a large and varying number of errors within the screen, the user may be overwhelmed, and may find it difficult

to isolate individual items of interest. In contrast, our implementation offers the ability to reveal classes of errors by selecting one or more categories, helping the user to focus her attention on each class of error in turn.

A second difference is that existing systems present grammar errors without any visual context for the recommended correction. As we have shown using the example of redundant phrases, our system offers the ability to tailor the visualization for a particular category of errors, to make it more effective for the user.

The conventional model of writing in word processors like Microsoft Word interleaves use of the grammar tools with the writing process, much like writing code in an integrated development environment. This flies in the face of a common approach to creative writing, which holds that it is important to separate the act of writing, which should be as free of distractions as possible, from the process of revision and improvement of what has already been written. For example, Boice reports that mixing creation with revision can lead to writers block, particularly in writers who tend towards perfectionism [5]. A tool designed to promote distraction-free writing would attempt to provide a composition environment that hides or removes features that are not essential to the task of writing itself.

A crop of such "distraction-free" editing tools have recently emerged. Examples are JDarkRoom [18], Q10 [4], and WriteRoom [15]. A sample screenshot of the editing environment Q10 is shown in Figure 6. MacIntyre describes these distraction-free environments as "zenware," noting that "unlike in Word, the choices are kept shrewdly off-screen: WriteRoom's blank slate reduces the urge to twiddle with margins and other formatting gewgaws. Instead, I find myself forgoing cosmetic changes for more functional ones, like bumping up the type size when my office window light starts to falter" [23].

Our approach, using ambient visualization, does not go as far as the proponents of distraction-free writing would like, in completely separating the creative writing phase from a later editing and revision phase. Instead, we offer a compromise between the paradigms of integrated writing and distraction-free writing. During the creative writing phase, Explanation View is hidden by default, allowing the

Figure 6: Distraction-free editing using Q10.

user to focus on generating content, although the Ambient View is still present. Should the user wish to switch to an editing and revision mode, she can interact with Stench Blossom to reveal any grammar issues of interest.

As with Stench Blossom's grammar advisor, other visualization tools have been developed to address various content generation and revision demands as related to writing activities. The VisRA readability visualization tool was built to help users identify and correct problems with English writing [35]. VisRA provides several visualizations, but the most detailed is similar to Stench Blossom's, in that they both depict problems as continuous variables and both allow users to see several types of problem at once. The main differences are that Stench Blossom is ambient instead of task-focused, and that it provides more detailed information about the rationale behind its advice through progressive disclosure. In contrast with the floss revising strategy of Stench Blossom, which focuses on the current

writing context, the Writing Blocks system is an alternative visualization tool that focuses on a more root canal revising approach through global activities that allow authors to visualize and understand the overall structure of documents [57]. The authors envision that global tools such as Writing Blocks can be combined with local tools like Stench Blossom, depending on type of revision needing to be performed. Lastly, Popout Prism is an overview and detail document interface designed to replace the traditional "find and highlight" technique used in document search [49]. Like Stench Blossom, Popout Prism utilizes perceptual principles to minimize distraction from the user's primary activity by removing the emphasis in Detail View (analogous to our Active View) when the user is not specifically performing a navigation task.

5.7 Evaluation

To evaluate our style and grammar advisor, we conducted a remote heuristic evaluation. We recruited evaluators through convenience sampling by contacting ten HCI experts in both academia and industry that were known to the authors, of which seven responded. We asked these seven evaluators to watch a two minute video demonstrating the usage of the grammar visualization. The video is based on the document and editing environment shown in Figure 4, and demonstrates typical interactions a user might perform when interacting with the advisor, such as scrolling through the document, and toggling one or more categories. In the video, the visualization overlapped the text at all times.

On a webpage, we provided evaluators with the following tool description:

> *For this evaluation, the general tool has been purposed for the task of grammar checking a LaTeX document within Eclipse. Although we have implemented our tool in Eclipse with LaTeX, the visualization technique is not specific to these technologies. The visualization is intended to be shown at all times, and to be used while creating and editing a document.*

The visualization is composed of sectors in a semicircle on the right-hand side of the editor pane. We call these sectors petals: each petal corresponds to a grammar category. The radius of the petal is proportional to the severity, so that an increased severity corresponds to an increased petal size.

Clicking on the petal toggles the highlighting of the grammar issues within the visible document region. For most categories, only simple highlighting is used, but more sophisticated highlighting is possible. For instance, for "Redundant Phrases," the entire redundant phrase is highlighted, but with the addition of a box surrounding the expression indicating the recommended expression to keep.

After watching the video, we asked the experts to complete Mankoff and colleagues' heuristic evaluation [25]. They were allowed to re-watch the video as needed during the process of completing the evaluation. We asked each evaluator to identify both the positive and negative aspects of the visualization for each heuristic. For the negative aspects of the heuristic, evaluators were additionally asked to indicate the severity of the issue from least severe (1) to most severe (5). To mitigate bias incurred as a result of the recruitment technique, we explicitly informed the evaluators that the primary purpose of a heuristic evaluation is to identify design problems, and that strong criticism was expected and encouraged.

5.7.1 Results

Mankoff and colleagues' evaluation covers eight different heuristics: (1) Useful and relevant information, (2) "Peripherality" of display, (3) Match between design of ambient display and environments, (4) Sufficient information design, (5) Consistent and intuitive mapping, (6) Easy transition to more in-depth information, (7) Visibility of state, and (8) Aesthetic and Pleasing Design [25]. We summarize each of these heuristics before presenting the evaluators' comments.

In presenting these results, the evaluators have been labeled as E1 through E7. When the evaluators identified a negative aspect, their assessment of the severity

of the problem is indicated in parentheses. When we omit a comment from an evaluator, either the evaluator did not provide a comment or made an irrelevant comment. For example, a few of the comments were related to either the Eclipse IDE or JLanguageTool. As another example, due to the way in which the video demo was presented, E1 and E3 were inadvertently led to believe that the visualization tool presents all outstanding issues within the document, rather than just the issues in the currently visible text; this led them to make irrelevant comments.

We made one change to the tool before making the demonstration video. Since the visualization is designed to be shown at all times, the toolbar icon that allows the user to disable and enable the tool was removed in the demonstration.

Useful and relevant information. Mankoff defines this heuristic as: *the information should be useful and relevant to the users in the intended setting*.

Positives. E1, E2, E4, and E7 indicated that presentating grammar issues in categories was a positive aspect of the design. E3 and E4 said that it was useful to be able to toggle information in one category on and off independently of other categories. E7 noted that selecting a category highlights the errors for that category.

Negatives. E1 indicated that to know where a category is, you have to interact with the visualization, or memorize its position (3). E2 indicated that the mapping between the color and the problem type was unclear (1), as was the metric used for severity (2). E2 was also concerned that only a limited number of categories can be presented (2). E3 commented that it is difficult to differentiate categories when multiple categories are activated (3). Though E2 and E4 indicated that categorization was a positive aspect of the visualization, they also noted that they are not sure how useful it is to classify grammatical issues by category (1, 3).

Peripherality of display. Heuristic: *the display should be unobtrusive and remain so unless it requires the user's attention. User should be able to easily monitor the display* [25].

Positives. E1, E3, and E5 indicated that the display is unobtrusive. E2 commented that it allows people to focus on their main task, which is to edit the document. E4, E5, and E6 mentioned the translucency as a positive aspect that aided unobtrusiveness.

Negatives. E1, E2, E5 and E6 indicated a problem in that the visualization overlaps the text (3, 3, 4, 2). While most evaluators did not provide a reason for why this is a problem, E2 stated that having the colored petals beneath the black text and surrounded by a white background makes it noticeable, and therefore affects the legibility. E2 and E4 questioned whether persistent display of the visualization was a good idea, particularly for distraction free writing (3, 3). E3 was unsure whether varying petal length is necessary for the visualization, since all errors must eventually be corrected anyway (2).

Match between design of ambient display and environments. Heuristic: *One should notice an ambient display because of a change in the data it is presenting and not because its design clashes with its environment*.

Positives. E1 indicated that the visualization does not change unless a new error is introduced in the current display, or unless you interact with it. E2, E5, and E6 mentioned that the colors are suited to the environment. E7 indicated that the design is ambient in that it falls into the background when reading the document.

Negatives. E1 believes that the typical use case for document editing is to correct issues as they arise. Consequently, the size of the petals would never increase to the point that they would become noticeable (4). Conversely, E1 and E7 indicated that if the petals grow too large, they are distracting (3, 2). E2, E3, and E5 all indicated that the overlay on existing text is distracting (1, 3, 4), an issue that was also identified in the display heuristic about peripherality. E5 added that some of the colors clash with the existing design of Eclipse (3), and E2 commented that the angles stand out when the document is scrolling, since most of the angles in

Eclipse are horizontal or vertical (2).

Sufficient information design. Heuristic: *The display should be designed to convey "just enough" information. Too much information cramps the display, and too little makes the display less useful*.

Positives. E2 mentions that it displays a nice summary, and E3 indicates that it is the "right mix of detail" versus giving an overview. E4 felt that the visualization allows you to focus on the task at hand, and E5 indicated that the information being displayed is useful. E6 observed that there was good use of labels and highlighting. E7 noted that the visualization makes users aware that they might have issues without dominating the screen space.

Negatives. E6 indicated that some of the petal segments are missing or very small, making them hard to activate; in fact, all petals, regardless of size, can be selected with equal ease, but this was not clear from the video. Furthermore, if a petal is not shown, a user may not even realize that the tool is monitoring that category (3). E7 thought it was confusing not to know how many problems exist in relation to the petal size (3). All other evaluators either indicated issues related to JLanguageTool, or evaluated this heuristic under the incorrect assumption that the visualization displays all errors in the entire document.

Consistent and intuitive mapping. Heuristic: *ambient displays should add minimal cognitive load. Cognitive load may be higher when users must remember what states or changes in the display mean. The display should be intuitive*.

Positives. E1, E3 and E4 provided no positive aspects for the design for this heuristic. E2 indicated that the severity "makes perfect sense" in that the visualization grows when there are a lot of issues, and shrinks when they are resolved; E5 and E7 made similar remarks. E6 felt that users would become used to the order of the categories over time.

Negatives. E1 and E4 intuitively felt that the petal should point to something in the document (5, 2). As a result of the demo, E1 perceived that there were more errors at the bottom of the document window than at the top, because of the way the petals were arranged. E2, E4, E5, and E6 all had criticisms about the lack of obvious meaning with respect to the colors of the petals (2, 2, 2, 3). E3 and E4 observed that the user needs to mouse-over the petals in order to remember what the categories are (2, 2). E6 observed that the highlighting of the other categories was light blue, but that "Redundant Phrases" used a different color for each error in the category (2). E5 commented that this is an unusual visualization, because most people think it is a pie, but it isn't, since only the radius, and not the central angle, matters. Experts may think it is a rose diagram, but unlike a rose diagram, the order of the petals does not indicate orientation within the document (5). E5 further suggests that a bar graph may be an alternative visualization.

Easy transition to more in-depth information. Heuristic: *if the display offers multi-leveled information, the display should make it easy and quick for users to find out more detailed information*.

Positives. E1 and E5 mentioned that the interaction is intuitive, with E2 indicating that on-demand information makes sense. E2 also liked the fact that the "active hover area" is larger than the petal itself. E3, E4, and E7 commented that clicking a petal, mousing over the petal, and selecting categories were positive aspects of meeting this heuristic. E6 liked the fact that multi-level information is available.

Negatives. E1 indicated that certain functionality might be difficult to access because because the colors of adjacent petals are difficult to distinguish (2). E2 and E6 were concerned that the user would accidentally click the document, and not the visualization (1, 3). E5 believes that it is not immediately obvious that the pie menu is clickable, since such menus are still not very common in user interfaces today (3).

Visibility of state. Heuristic: *an ambient display should make the states of the system noticeable. The transition from one state to another should be easily perceptible*.

Positives. E1 and E2 indicated that there are very clear transitions between states, and that it is easy to tell what state the tool is in. E4 indicated that the mouse hover actions change only one category at a time. E5 and E7 indicated that the petal growth is a positive indicator of the visibility of state.

Negatives. E3 was not sure how to tell when a category was active or inactive (2), and E4 noticed that there is no way to see all of the errors at once without toggling each category petal one by one (3). E5 identified that the constantly changing petals could be quite distracting (3). E6 noted that, depending on position, the mouse rollover labels can obscure the problem that is being highlighted (4). E6 also observed that the darkness of some petals can obscure the text, which is yet another indication that it can be distracting for the visualization to overlay the text (1). E7 was unsure of the rate at which the petals grow (2).

Aesthetic and pleasing design. Heuristic: *the display should be pleasing when it is placed in the intended setting*.

Positives. E1, E2, E3, E4, E5 identified the color scheme as being generally pleasing, with E3 indicating that the colors had low saturation, low contrast from the background, but were still visibly different. E7 indicated that the transparent display had a "good feel to it" and was well placed.

Negatives. E1 commented that the visualization is cluttered where it overlaps with the text, making it busy looking (2). E2 mentions that the angles are a bit jarring given the boxy look and feel of the document editor (1). E5 was unsure if users would be pleased with the visualization constantly changing as they type (3).

5.7.2 Evaluation Summary

An examination of the heuristic evaluation results reveals a set of issues that were commonly identified by many of the evaluators. A recurring issue with respect to color is that, while the use of pastels are aesthetically pleasing, the colors should encode some form of intuitive meaning. Although we intended to have the blue-to-orange gradients signify less-obvious to more-obvious problems, this was not apparent to the evaluators. This problem exists despite the fact that the colors are not strictly necessary to interpret the visualization.

Many of the evaluators also criticized the visualization for overlapping the text. This is a problem that was not so pronounced in code, because code typically has hard line breaks and does not extend to the right-hand margin, whereas English text is usually word wrapped, and so occupies the full width of the editing pane. This criticism suggests that users may want a separation between the visualization and the text.

A number of other issues merit attention, because they were identified by individual evaluators as very severe. In "Match between design of ambient display and environments", E1 believes that the "typical use case" of document editing is to correct issues as they arise, which suggests a behavior that is reminiscent of "floss refactoring" for code smells. In such a scenario, the petals would likely remain small and hard to notice, and therefore a binary representation (on or off) may be more valuable than a continuous representation for grammar issues. In contrast, E2 indicates that he "like[s] to get all the ideas down, not worrying about little issues like grammar", which can be fixed in later iterations. In such a case, grammar issues would continue to accrue until the evaluator addressed them. We can infer that this evaluator seems to prefer a "root canal refactoring" approach to tackling grammar issues. The dichotomy is interesting because the Stench Blossom's design originated from our desire to support floss refactoring. The received wisdom for code is that flossing is good; this is not so clear for writing.

In the evaluation, E5 explicitly stated that the visualization can be confusing because it is a hybrid between a pie chart and a rose diagram. Implicitly, E1 and E4

mentioned the same issue, because they felt that the petals should point to something; this supports the idea that the use of petals may be confusing. In the broader context, E2 mentioned the general idea of angles as being distracting, because of the horizontal and vertical angles of most user interfaces, including Eclipse. E5's suggestion that the visualization should instead represent issues as a bar graph would solve this particular problem.

Some of the identified issues can be used to improve the tool, since they can be implemented by making small changes that do not conflict with Stench Blossom's overall design. For instance, E3 was not sure when a category was active, an issue that can resolved by rendering the active petals at full saturation whenever their explanation view is displayed. As for E1's suggestion that users will not allow more than a few issues to accumulate at any one time, the function controlling petal size could also be adjusted, so that even a small number of issues for a category quickly amplifies the size of the petal.

Many of the evaluators' criticisms may stem from the fact that they may have unconsciously been comparing Stench Blossom against existing grammar tools that use the "wavy underlining" user interface. For instance, some evaluators questioned whether or not categorizing issues is useful in the first place. This is an unavoidable shortcoming of this style of evaluation: a new visualization may be criticized for being "different", even though these differences may offer affordances for addressing style and grammar issues that are not available in existing tools. For example, existing grammar checking tools may simply omit the presentation of subjective issues (such as starting a sentence with a conjunction) or frequently occurring issues (such as passive voice), to avoid overwhelming the user. A benefit of Stench Blossom's "difference" is that it offers a means to display a greater range of style and grammar issues, because the user can choose to view them selectively, or not at all.

6 Future Work

We feel that future work on Stench Blossom could proceed in at least three directions. First, the Stench Blossom smell detector can be improved in several ways. Second, the Stench Blossom grammar tool can also be improved. Third, the concepts used in the design of Stench Blossom may be beneficial in other areas.

6.1 Improvements to Stench Blossom

As we discussed in Section 4, subjects sometimes did not agree with the tool's estimate of the strength of a code smell. One way to deal with this would be to allow the programmer to drag the edges of the petals toward or away from the center of the visualization, so that the visualization more closely matches the intuition of the developer. This would provide a convenient way for developers to specify individual preferences so that the tool can adapt to those preferences in the future. Likewise, if developers do not agree with our ranking of the obviousness of smells, then the visualization could allow the developer to change the petals' vertical ordering by dragging the petals.

Some subjects in the evaluation suggested that there were fundamental differences in the granularity of the smells, and displaying them uniformly was confusing. Specifically, LARGE CLASS was at the class level, while the other smells were at the method level. In future versions of Stench Blossom, making a visual distinction between the different levels of granularity may help programmers understand the visualization more quickly.

Another possible modification to Stench Blossom would be to display information about which smells are increasing or decreasing as a programmer is coding, rather than displaying information about the code as it is now (we thank Bill Pugh for this suggestion). Using this information, the programmer would be made aware of the effect that her changes are having on the smelliness of the code.

6.2 Grammar Improvements to Stench Blossom

The addition of more elaborate rules in the JLanguageTool library would provide an avenue for more advanced visualizations. Since most rules are based on patterns, grammar issues in JLanguageTool are detected locally rather than globally. For instance, a possible bad style issue not correctly detected is that of mixed verb tenses within a document. One visualization could color verbs from one tense in a color, and verbs from another tense in a different color. The user could then visually determining the weighting between the two verb tenses and make an appropriate decision.

A current limitation of the Stench Blossom system is that there is no assisted correction. For rules that require minimal context, such as overregularized verb inflection (`goed` vs. `went`) , the system should provide a user interface to quickly correct the issue. Right-clicking on the visualized word and selecting an appropriate correction from a drop down menu is one possible solution. For redundant phrases, the user might opt to double-click on the expression they wish to keep, telling the system to automatically remove the other.

6.3 Further Applications

We feel that the visualization technique and guidelines that we have presented in this paper are useful beyond the code smell and grammar tools that we have described. Based on our experience implementing Stench Blossom in these two domains, we believe that it is useful for visualizing information that has the following properties:

- acting on the information is not an urgent priority;
- users interpret the information subjectively;
- the information relates to complex properties of artifacts;
- the information relates to user-changeable properties of artifacts, but users are unlikely to completely eliminate those properties; and

- the information is useful primarily in the context of working with existing artifacts for tasks loosely related to that information.

These properties may hold for other tools used in software development and for soft advice systems in other domains.

As an example of another software development tool that might benefit from our guidelines, consider Ensemble, a system that recommends collaborators to software developers, based on the work that they are doing [55]. Ensemble might notice that Ira is working in a method, recognize that three other developers are currently working on similar methods, and then recommend that Ira collaborate with those developers. Generalizing our Estimability guideline suggests that Ensemble should give an approximation of how much effort would be required if Ira took Ensemble's advice and collaborated with one of the other developers. For instance, the tool might tell Ira that Jan is working from home today (high effort to collaborate), Kim has a meeting in a few minutes (medium effort), and Lou has an open schedule and is sitting in close proximity to Ira (low effort).

Our experience in applying our visualization technique to a grammar advisor has widened our view of what interactive ambient visualizations can be used for. For example, although our code smell detector and grammar advisor both work over text-based artifacts, our visualization technique may be useful for other artifacts such as graphic design documents. In such documents, a graphic designer may be using a program like Adobe Illustrator or CorelDRAW to creatively compose a graphical depiction for an intended audience, such as the logo for a business or a poster for a conference. Like programming and writing, graphic design has numerous soft guidelines that produce quality results when followed, but also can produce quality results when expertly disobeyed. Examples of such guidelines include "use one or two odd shapes and make the rest regular shapes" and "use the same color palette throughout" [1]. Standard highlighting techniques where the graphic design document is constantly analyzed for guideline violations and violations are immediately shown as highlights on the document (similar to text underlines) is especially inappropriate in the graphic design setting; such highlights

would immediately pollute and distort the document's aesthetic value. Instead, we hypothesize that using interactive ambient visualizations can provide immediate feedback to graphic designers without diminishing the aesthetic experience.

7 Conclusion

Tools that offer soft advice can help the users of a software system to improve the quality of their work. However, soft advice should ideally be delivered in a way that is deeply explanatory, that does not overwhelm the user, that is not obtrusive or distracting, that is sensitive to the user's working context, that is easily accessible at all times, that depicts relationships between artifacts, that emphasizes difficult-to-identify issues, and that helps the user to estimate the extent of the underlying problem. Our experience building, using, and evaluating Stench Blossom, in the context of both code smells and grammar issues, has allowed us to reflect on why these ideals are important and how they can be achieved. We hope that this research has helped to clarify the role of tools that present soft advice: not front and center stage, where they get in the way of the primary task, but in the background, always ready to offer advice on when requested, but keeping a low profile when the user is focused on other tasks.

References

[1] The basics of graphic design. Internet. http://www.online.tusc.k12.al.us/tutorials/grdesign/grdesign.htm.

[2] E.S. Al-Shaer and H.H. Hamed. Firewall policy advisor for anomaly discovery and rule editing. In *Integrated Network Management, 2003. IFIP/IEEE Eighth International Symposium on*, pages 17 – 30, march 2003.

[3] Apache. Tomcat, 2012. Computer Program, http://tomcat.apache.org/.

[4] Joaqun Bernal. Q10, 2012. Computer Program, http://www.baara.com/q10/.

[5] Robert Boice. *Professors as Writers*. New Forum Press, 1990.

[6] J. Callahan, D. Hopkins, M. Weiser, and B. Shneiderman. An empirical comparison of pie vs. linear menus. In *CHI '88: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 95–100. ACM, 1988.

[7] René Coppieters. Competence differences between native and near-native speakers. *Language*, 63(3):pp. 544–573, 1987.

[8] Tracey M. Derwing, Marian J. Rossiter, and Maureen Ehrensberger-Dow. 'they speaked and wrote real good': Judgements of non-native and native grammar. *Language Awareness*, 11(2):84 – 99, 2002.

[9] M. Drozdz, D. G. Kourie, B. W. Watson, and A. Boake. Refactoring tools and complementary techniques. In *AICCSA '06: Proceedings of the IEEE International Conference on Computer Systems and Applications*, pages 685–688. IEEE Computer Society, 2006.

[10] Eva van Emden and Leon Moonen. Java quality assurance by detecting code smells. In *Proceedings of the Ninth Working Conference on Reverse Engineering*, pages 97–106. IEEE Computer Society, 2002.

[11] Jill Fitzgerald. Research on Revision in Writing. *Review of Educational Research*, 57(4):481–506, December 1987.

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

[13] Michael Gamon. Using mostly native data to correct errors in learners' writing: a meta-classifier approach. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, HLT '10, pages 163–171, Stroudsburg, PA, USA, 2010. Association for Computational Linguistics.

[14] Jennifer Gluck, Andrea Bunt, and Joanna McGrenere. Matching attentional draw with utility in interruption. In *CHI '07: Proceedings of the SIGCHI*

conference on Human factors in computing systems, pages 41–50, New York, NY, USA, 2007. ACM.

[15] Jesse Grosjean. WriteRoom, 2012. Computer Program, http://www.hogbaysoftware.com/products/writeroom.

[16] Na-Rae Han, Martin Chodorow, and Claudia Leacock. Detecting errors in English article usage by non-native speakers. *Natural Language Engineering*, 12(02):115–129, 2006.

[17] Shinpei Hayashi, Motoshi Saeki, and Masahito Kurihara. Supporting refactoring activities using histories of program modification. *IEICE - Transactions on Information and Systems*, E89-D(4):1403–1412, 2006.

[18] Duncan Jauncey. JDarkRoom, 2012. Computer Program, http://www.codealchemists.com/jdarkroom.

[19] Kimmo Karlsson. TeXlipse, 2012. Computer Program, http://texlipse.sourceforge.net.

[20] Mik Kersten and Gail 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. ACM, 2005.

[21] Rainer Koschke, Christopher D. Hundhausen, and Alexandru Telea, editors. *Proceedings of the ACM 2008 Symposium on Software Visualization, Ammersee, Germany, September 16–17, 2008*. ACM, 2008.

[22] Hee-Kyung Lee. Native and nonnative rater behavior in grading korean students' english essays. *Asia Pacific Education Review*, 10:387–397, 2009. 10.1007/s12564-009-9030-3.

[23] Jeffrey MacIntyre. The Tao of Screen, 2012. Computer Program, http://www.slate.com/id/2182744.

[24] Pattie Maes. Agents that reduce work and information overload. *Communications of the ACM*, 37(7):30–40, 1994.

[25] Jennifer Mankoff, Anind K. Dey, Gary Hsieh, Julie Kientz, Scott Lederer, and Morgan Ames. Heuristic evaluation of ambient displays. In *CHI '03:*

Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pages 169–176. ACM, 2003.

[26] Mika 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, November 2005.

[27] Mika V. Mäntylä, Jari Vanhanen, and Casper Lassenius. Bad smells – humans as code critics. *IEEE International Conference on Software Maintenance*, 0:399–408, 2004.

[28] Daniel McFarlane. Comparison of four primary methods for coordinating the interruption of people in human-computer interaction. *Hum.-Comput. Interact.*, 17(1):63–139, 2002.

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

[30] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. How we refactor, and how we know it. In *ICSE '09: Proceedings of the 31st International Conference on Software Engineering*, 2009.

[31] Daniel Naber. A rule-based style and grammar checker. Diploma thesis, August 2003. Technische Fakultat and Universitat Bielefeld.

[32] Helmut Neukirchen and Martin Bisanz. Utilising Code Smells to Detect Quality Problems in TTCN-3 Test Suites. In *Proceedings of the 19th IFIP International Conference on Testing of Communicating Systems and 7th International Workshop on Formal Approaches to Testing of Software*, pages 228–243. Springer, Heidelberg, June 2007.

[33] Jakob Nielsen. Ten usability heuristics. Internet, 2005. http://www.useit.com/papers/heuristic/heuristic_list.html.

[34] Jakob Nielsen and Rolf Molich. Heuristic evaluation of user interfaces. In *CHI '90: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 249–256, New York, NY, USA, 1990. ACM.

[35] D. Oelke, D. Spretke, A. Stoffel, and D.A. Keim. Visual readability analysis: How to make your writings easier to read. In *Visual Analytics Science and Technology (VAST), 2010 IEEE Symposium on*, pages 123 –130, oct. 2010.

[36] Oracle. OpenJDK, 2012. Computer Program, http://openjdk.java.net/groups/core-libs.

[37] Chris Parnin and Carsten Görg. Building usage contexts during program comprehension. In *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension*, pages 13–22. IEEE Computer Society, 2006.

[38] Chris Parnin and Carsten Görg. Design guidelines for ambient software visualization in the workplace. *Visualizing Software for Understanding and Analysis, 2007. VISSOFT 2007. 4th IEEE International Workshop on*, pages 18–25, June 2007.

[39] Chris Parnin, Carsten Görg, and Ogechi Nnadi. A catalogue of lightweight visualizations to support code smell inspection. In Koschke et al. [21], pages 77–86.

[40] Zachary Pousman and John Stasko. A taxonomy of ambient information systems: four patterns of design. In *Proceedings of the working conference on Advanced visual interfaces*, AVI '06, pages 67–74, New York, NY, USA, 2006. ACM.

[41] Ananda Rakshit, Nirup Krishnamurthy, and Gang Yu. System operations advisor: A real-time decision support system for managing airline operations at united airlines. *INTERFACES*, 26(2):50–58, 1996.

[42] Jef Raskin. *The humane interface: new directions for designing interactive systems*. ACM Press/Addison-Wesley Publishing Co., 2000.

[43] T. J. Robertson, Shrinu Prabhakararao, Margaret Burnett, Curtis Cook, Joseph R. Ruthruff, Laura Beckwith, and Amit Phalgune. Impact of interruption style on end-user debugging. In *CHI '04: Proceedings of the SIGCHI*

conference on Human factors in computing systems, pages 287–294, New York, NY, USA, 2004. ACM.

[44] Ben Shneiderman. System message design: Guidelines and experimental results. In Albert Badre and Ben Shneiderman, editors, *Directions in Human/Computer Interaction*, Human/Computer Interaction, chapter 3, pages 55–78. Ablex Publishing Corporation, 1982.

[45] Ben Shneiderman. *Designing the User Interface (2nd ed.): Strategies for Effective Human-Computer Interaction*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[46] Frank Simon, Frank Steinbrückner, and Claus Lewerentz. Metrics based refactoring. In *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering*, pages 30–38. IEEE Computer Society, 2001.

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

[48] Azureus Software. Vuze, 2012. Computer Program, http://vuze.com.

[49] Bongwon Suh, Allison Woodruff, Ruth Rosenholtz, and Alyssa Glass. Popout prism. In *Proceedings of the SIGCHI conference on Human factors in computing systems Changing our world, changing ourselves - CHI '02*, page 251, New York, New York, USA, April 2002. ACM Press.

[50] Anne M. Treisman and Garry Gelade. A feature-integration theory of attention. *Cognitive Psychology*, 12(1):97–136, January 1980.

[51] Nikolaos Tsantalis, Theodoros Chaikalis, and Alexander Chatzigeorgiou. JDeodorant: Identification and removal of type-checking bad smells. In *CSMR*, pages 329–331. IEEE Computing Society, 2008.

[52] G. Valentin, M. Zuliani, D.C. Zilio, G. Lohman, and A. Skelley. Db2 advisor: an optimizer smart enough to recommend its own indexes. In *Data Engineering, 2000. Proceedings. 16th International Conference on*, pages 101 –110, 2000.

[53] Maarten van Dantzich, Daniel Robbins, Eric Horvitz, and Mary Czerwinski. Scope: providing awareness of multiple notifications at a glance. In *Proceedings of the Working Conference on Advanced Visual Interfaces*, AVI '02, pages 267–281, New York, NY, USA, 2002. ACM.

[54] Richard Wettel and Michele Lanza. Visually localizing design problems with disharmony maps. In Koschke et al. [21], pages 155–164.

[55] P. F. Xiang, A. T. T. Ying, P. Cheng, Y. B. Dang, K. Ehrlich, M. E. Helander, P. M. Matchen, A. Empere, P. L. Tarr, C. Williams, and S. X. Yang. Ensemble: a recommendation tool for promoting communication in software teams. In *RSSE '08: Proceedings of the 2008 International Workshop on Recommendation Systems for Software Engineering*. ACM, 2008.

[56] Rebecca Xiong and Judith Donath. PeopleGarden: creating data portraits for users. In *Proceedings of the 12th annual ACM symposium on User interface software and technology*, UIST '99, pages 37–44, New York, NY, USA, 1999. ACM.

[57] Sheng Xu and Hirohito Shibata. Writing blocks: a visualization to support global revising. In *Proceedings of the 2007 conference of the computer-human interaction special interest group (CHISIG) of Australia on Computer-human interaction: design: activities, artifacts and environments - OZCHI '07*, page 61, New York, New York, USA, November 2007. ACM Press.

Appendix

On the following pages, you will find the experimenter's notebook used in the experiment, including the pre-test questionnaire, experiment administrator's guide, post-test questionnaire, and $3'' \times 5''$ code smell cards.

Pre-Experiment Questionnaire

The following questionnaire is intended for us to get an idea of what your programming background is. Your answers in no way affect the rest of the experiment, it simply gives us context for interpreting the result.

Feel free to write in the margins to explain your answers, if necessary.

Job title: _____

How many years have you been programming? _____

Over the last year, about how many hours per week would you say you spend programming, on average? _____

How proficient, on a scale from 0 to 4, where 0 means "not at all" and 4 means "expert"?

Java	*0*	*1*	*2*	*3*	*4*
C++	*0*	*1*	*2*	*3*	*4*

When programming, do you typically use an Integrated Development Environment? *Y / N*
If so, which one(s) and for what % time? _____
What non-IDE editors do you use for programming? _____

On a scale form 0 to 4, how familiar are you with the practice of refactoring?
(0 = not at all, to 4 = very familiar) *0* *1* *2* *3* *4*

Do you use any refactoring tools? *Y / N* If so, which ones?

_____

On a scale form 0 to 4, how familiar are you with code smells?
(0 = not at all, to 4 = very familiar) *0* *1* *2* *3* *4*

Please Hand This Back to Experimenter

Experimental Procedure

Introduction

What we're going to do in this experiment is investigate code smells, which were originally proposed in Martin Fowler's book on refactoring. The idea is that smells help you identify candidates for refactoring; for instance, the "Long Method" smell suggests that you should perhaps perform the Extract Method refactoring. You needn't be too familiar with the concept; we'll do some review as we go along and you are free to ask questions.

This experiment will have four parts:

[AB] In the first part, I'll ask you about smells in code. In the second part, I'll give you a tool to help find smells.
In the third part, I'll ask you some details about smells, and in the fourth part, I'll ask you about the details with the assistance of the tool.

[BA] In the first part, I'll ask give you a tool to help find smells. In the second part, I'll ask you about smells without the help of a tool.
In the third part, I'll ask you some details about smells with the help of the tool, and in the fourth part, I'll ask you about the details without the tool.

In a moment, I'll give you eight 3 by 5 cards. Each card will have the name of a code smell and its definition, with an example on the back. I'll give you a few minutes to read them, then you'll give them back to me when you're ready, and then we'll begin looking at some code. Questions?

Give the participant the card stack, and await their completion. If it takes more than a few minutes, ask them if they are finished. If they are not satisfied within 10 minutes, tell them that we'll move on regardless, and that you'll make a note that they were not finished.

Ok, so now we're going to look at some code. As you work, please don't modify the code, or navigate outside of the editor. As a rule of thumb, please try to spend no more than 3-5 minutes per file.

[A] Manual Finding

Take cards back. Now I'll ask you to look at a Java file and try to spot some of the code smells that you saw on the cards. You'll scroll through one Java file, while skimming the code top to bottom. If you see an interesting smell, just say so out loud.

Open up ToolDemo. So for instance, you would scroll through this file from top to bottom, noting any smells you notice.

Questions?

[1] Open scroll1.		**[2]** Open scroll3.
Data Clumps Feature Envy Message Chains Switch Statements Typecast Instanceof Large Method Large Class		
[1] Open scroll8.		**[2]** Open scroll7.
Data Clumps Feature Envy Message Chains Switch Statements Typecast Instanceof Large Method Large Class		

[B] Tool Finding

Take cards back. I'll ask you to spot some of the code smells that you saw on the cards. You'll scroll through one Java file, while skimming the code top to bottom, with the help of a smell detection tool.

(Open up ToolDemo, activate tool) The tool is represented by a visualization behind your java code. *(Scroll).* It looks a bit like a bunch of petals on a flower. Each petal represents a smell, and we can hover over to see the name of the smell *(demo)*. The size of the petal represents how bad that smell is in the code that you are looking at. As this tripwire passes over methods *(demo)*, or when the cursor is in a method, the smells for that method are visualized.

This part of the tool is intended to give you an idea of which smells are present. There's more detail to the tool, but we'll get to that later.

So, the task is, if the tool helps you see an interesting smell, just say so out loud. Ready?

[1] Open scroll3.		**[2]** Open scroll1.
Data Clumps Feature Envy Message Chains Switch Statements Typecast Instanceof Large Method Large Class		
[1] Open scroll7.		**[2]** Open scroll8.
Data Clumps Feature Envy Message Chains Switch Statements Typecast Instanceof Large Method Large Class		

[A] Manual Finding

(Switch to full screen editor)
Now what we'll do is look at one code smell in depth; Feature Envy. *(Open up ToolDemo)* Suppose I analyze this for Feature Envy by inspection.

Looking at this detail, I might conclude that the method, or some parts of it, should be moved to DHTTransportFullStats.

So the task that I want you to do is to make some judgments about the code; how widespread the Feature Envy is, how likely you are to remove it, and how you might do it. I'll ask these questions as you work, and if you have any questions for me, feel free to ask. When you're finished, let me know.

Any questions? *(Pause)* Ok, give it a try on this method.

[1] Open envy8.	**[2]** Open envy3.
How widespread is the smell? How likely are you to remove it? How might you remove it?	
[1] Open envy1.	**[2]** Open envy6.
How widespread is the smell? How likely are you to remove it? How might you remove it?	

[B] Tool Inspection

(Switch to full screen editor)
Now what we'll do is look at one code smell in depth; Feature Envy. *(Open up ToolDemo)* Suppose that I glance at the smell indicator and see that Feature Envy is high. I can then click on its label *(do it)*, and get a detailed view of what's going on.

The movable sheet shows me which classes members are referenced, and assigns each class a color. So for instance *(point)*, I can see that many members of DHTTransportFullStats are referenced, but only one member in this class is referenced. The associated members are highlighted in source code, and I can mouse-over the classes and members to emphasize their occurrences in code.

Looking at this detail, I might conclude that the method, or some parts of it, should be moved to DHTTransportFullStats.

So the task that I want you to do is to use the tool to help you make some judgments about the code; how widespread the Feature Envy is, how likely you are to remove it, and how you might do it. I'll ask these questions as you work, and if you have any questions for me, feel free to ask. When you're finished, let me know.

Any questions? *(Pause)* Ok, give it a try on this method.

[1] Open envy3.	**[2]** Open envy8.
How widespread is the smell? How likely are you to remove it? How might you remove it?	
[1] Open envy6.	**[2]** Open envy1.
How widespread is the smell? How likely are you to remove it? How might you remove it?	

Post-Experiment Questionnaire

Please answer a few questions about your experience during this experiment. Feel free to write comments in the margins. We just saw a tool to assist in the detection and understanding of code smells. Below are two questions about several characteristics that such a tool might have.

Characteristic	How important is the characteristic to *any smell detection tool*?						Do you agree that the characteristic *applies to the tool you just used*?				
	Not Important	Somewhat Important	Important	Very Important	Essential		Strongly Disagree	Somewhat Disagree	Neutral	Somewhat Agree	Strongly Agree
Example: The tool should help me cook dinner.	☐	☐	☐	☒	☐		☒	☐	☐	☐	☐
Example: The tool should use pretty colors.	☒	☐	☐	☐	☐		☐	☐	☐	☒	☐
The tool should not distract me.	☐	☐	☐	☐	☐		☐	☐	☐	☐	☐
The tool should have a user interface consistent with the rest of the environment.	☐	☐	☐	☐	☐		☐	☐	☐	☐	☐
The tool should make smell information available to me at all times.	☐	☐	☐	☐	☐		☐	☐	☐	☐	☐
The tool should tell me first and foremost about smells related to the code I'm working on.	☐	☐	☐	☐	☐		☐	☐	☐	☐	☐
The tool should not block me from my other work while it analyzes or finds smells.	☐	☐	☐	☐	☐		☐	☐	☐	☐	☐
The tool should emphasize smells that are difficult to see with the naked eye.	☐	☐	☐	☐	☐		☐	☐	☐	☐	☐
The tool should not overwhelm me with the smells that it detects.	☐	☐	☐	☐	☐		☐	☐	☐	☐	☐
In addition to finding smells for me, the tool should tell me *why* smells exist.	☐	☐	☐	☐	☐		☐	☐	☐	☐	☐

When showing me details about code smells, the tool should show me the relationships between effected program elements.	☐ ☐ ☐ ☐ ☐		☐ ☐ ☐ ☐ ☐
The tool should help me estimate the extent of a smell in the code.	☐ ☐ ☐ ☐ ☐		☐ ☐ ☐ ☐ ☐
The tool should help me decide whether to remove a smell from the code.	☐ ☐ ☐ ☐ ☐		☐ ☐ ☐ ☐ ☐

Please state whether you agree with the following statements:

	Strongly Disagree	Somewhat Disagree	Neutral	Somewhat Agree	Strongly Agree
The smell detector that I used in this experiment was *useful for the given tasks*	☐	☐	☐	☐	☐
The detector found information that I would not have found *as quickly* without it.	☐	☐	☐	☐	☐
The detector found information that I would not have found *at all* without it.	☐	☐	☐	☐	☐
Without the tool, it was difficult to look for all 8 smells at the *same time*.	☐	☐	☐	☐	☐
If a detector like the one in this experiment were available, *I would use it* when I code.	☐	☐	☐	☐	☐

Please Hand This Back to Experimenter

Loose Interview

If you were using this tool while coding, do you think that it would get your attention at the right times?

Would it be too distracting?

Did the tool make you more confident about your refactoring judgements, with respect to feature envy?

Do you think it helped you make more informed judgments?

If you could *change something* about the smell detector, what would it be?

Data Clumps

Data clumps are multiple parameters that appear repeatedly in several method signatures.

Data clumps are a problem because a change to one clump necessitates the same change in several places.

To correct this problem, the duplicated parameters should be encapsulated into objects.

For instance, if the integers x, y, and z appear as parameters to several methods, they could be replaced by a new 3DPoint class.

Feature Envy

Feature envy occurs when code relies heavily on data and methods from other classes, more that those from its own class.

Feature envy is a problem because the responsibilities of a class should be contained in the class itself.

To correct this problem, code that exhibits feature envy should be moved into the appropriate class.

Message Chain

A message chain is a long series of method calls on the same object.

Message chains are a problem because the client code becomes tightly coupled with the object's structure.

To correct this problem, parts of the message chains can be hidden in various objects, or methods can be moved closer to clients.

For instance, this.getFigure().getPolygon(0).getPoint(0).x() might be replaced by this.farthestLeft().

Switch Statement

A switch statement is a language feature.

Switch statements are a problem because, often, they are duplicated many places in code.

To correct this problem, switch statements may be replaced with polymorphism.

Feature Envy: Example

```
class Figure{
void add(Point p1, Point p2){
   return new Point(p1.x+p2.x,p1.y+p2.y);
}
…
```

Data Clumps: Example

```
void add(int x, int y, int z){…}
void remove(int x, int y, int z){…}
void shift(int x, int y, int z){…}
void with(int x, int y, int z){…}
```

Switch Statement: Example

```
String pointString(){
 switch(getPoint().getType()){
  case TWO_DIMENSIONAL: return “2d”;
  case THREE_DIMENSIONAL: return “3d”;
  case FOUR_DIMENSIONAL: return “4d”;
  default: return “unknown”;}
}
```

Message Chain: Example

```
this.getFigure().getPolygon(0).getPoint(0).x()
```

Typecast

A typecast is where an object of one type is cast to an object of another type.

Typecasts are a problem because illegal casts can be written that will result in runtime errors.

Typecasts can sometimes be replaced with generics.

InstanceOf

An InstanceOf is a language feature.

InstanceOf is a problem because they are often duplicated many places in code.

InstanceOf checks can sometimes be replaced with polymorphism.

Large Method

A large method is a method that is long or does too many things.

Large methods are a problem because they can be difficult to understand.

Large methods can often be broken down into smaller methods.

Large Class

A large class is a class that is too long or has too many responsibilities.

Large classes are a problem because they can be difficult to understand.

Large classes can often be broken down into smaller classes.

InstanceOf: Example

```
String pointString(Point p){
 if(p instanceof 2DPoint){
   return “2d”;
 }else if(p instanceof 3DPoint){
   return “3d”;
 }else if(p instanceof 4DPoint){
   return “4d”;
 }else{
   return “unknown”;}
}
```


Typecast: Example

```
void add(Collection c){
  this.x = ((Integer)c.get(0)).intValue():
  this.y = ((Integer)c.get(1)).intValue():
  this.z = ((Integer)c.get(2)).intValue():
}
```

Large Class: Example

(some class that does not fit on this card)

Large Method: Example

(some method that does not fit on this card)