An Interactive Ambient Visualization for Code Smells

Emerson Murphy-Hill
Department of Computer Science
North Carolina State University
Raleigh, North Carolina
emerson@csc.ncsu.edu

Andrew P. Black
Department of Computer Science
Portland State University
Portland, Oregon
black@cs.pdx.edu

ABSTRACT

Code smells are characteristics of software that indicate that code may have a design problem. Code smells have been proposed as a way for programmers to recognize the need for restructuring their software. Because code smells can go unnoticed while programmers are working, tools called smell detectors have been developed to alert programmers to the presence of smells in their code, and to help them understand the cause of those smells. In this paper, we propose a novel smell detector called Stench Blossom that provides an interactive ambient visualization designed to first give programmers a quick, high-level overview of the smells in their code, and then, if they wish, to help in understanding the sources of those code smells. We also describe a laboratory experiment with 12 programmers that tests several hypotheses about our tool. Our findings suggest that programmers can use our tool effectively to identify smells and to make refactoring judgements. This is partly because the tool serves as a memory aid, and partly because it is more reliable and easier to use than heuristics for analyzing smells.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques; H.5.2 [**Information interfaces and presentation**]: Graphical user interfaces

General Terms

Design, Human Factors

Keywords

Software, refactoring, usability, code smells

1. INTRODUCTION

Liliana is a hypothetical programmer working on the Apache Tomcat project (http://tomcat.apache.org/). 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" [4]: 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 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 [4]; other researchers (for example, van Emden and Moonen [3]) 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 [12]. 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

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.

Table 1: Some Java Code Smells identified by Fowler [4], by van Emden and Moonen [3] (indicated with *), and by Drozdz and colleagues [2] (indicated 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 she could have used to add features or fix bigs. Without setting aside time specifically for finding code smells, programmers may not notice them.

For these two 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 smell analysis to the programmer using the tool.

While there has been significant research on the code analysis portion of smell detectors, there has been relatively little work on the design and evaluation of user interfaces for smell detectors. That will be the primary focus of this paper: our main contribution is the design and evaluation of a user-interface for visualizing code smells.

This paper describes a novel smell detector called Stench Blos-
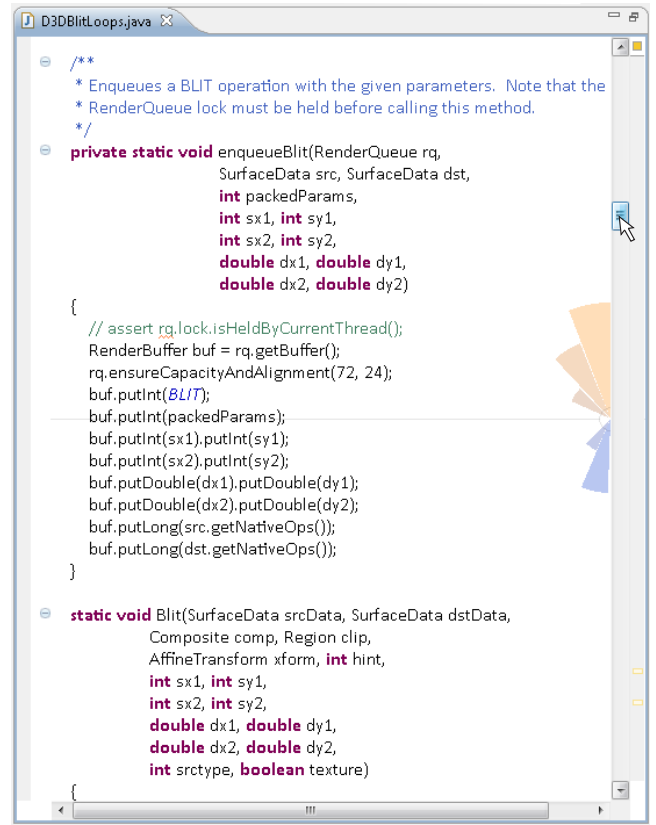
Figure 1: Ambient View in Stench Blossom.

som that uses an interactive ambient visualization (Section 2). 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 (Section 3). We also describe an experiment to evaluate several hypotheses about Stench Blossom and its associated guidelines (Section 4). In Section 5 we describe some potential future improvements for Stench Blossom and speculate that the lessons learned from its design can be applied to tools beyond smell detectors.

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 http://multiview.cs.pdx.edu/refactoring/smells.

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.
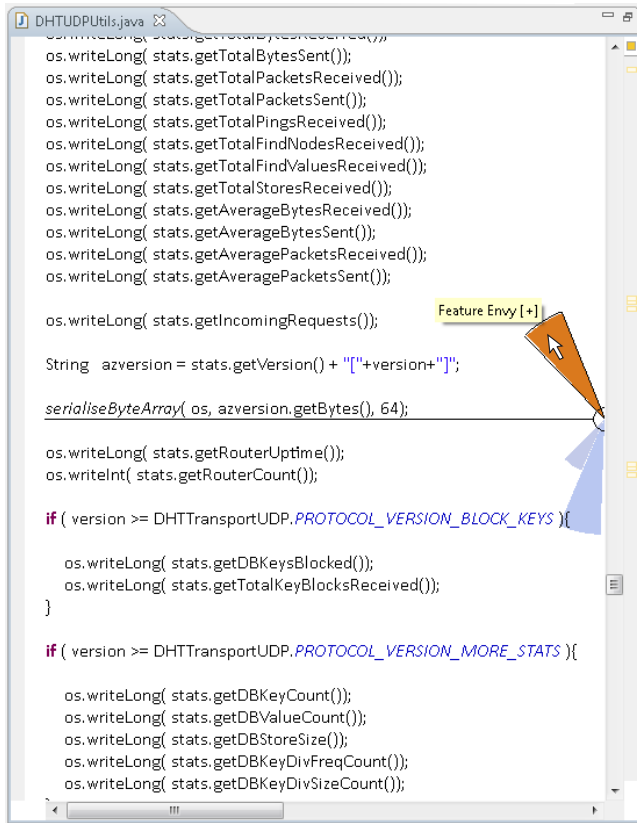
Figure 2: Active View in Stench Blossom.

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 [15]. This design choice is in contrast to that made by smell detectors that must be explicitly invoked to view their results, such as Crocodile [26]; 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 [16].

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 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 [3] or CodeCity [30]; 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" [11].

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 the 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. We kept the visualization simple, rather than displaying, for example, the names of the smells, in order to avoid "information overload" [10]. 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 [1]. For example, a programmer may learn that FEATURE ENVY always appears in the ↖ 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.

The radius of the petal represents the strength of the smell, where a radius of zero represents an absence of the smell, up to the full radius, representing 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 [17] and CodeNose [27], 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 [14]. 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 [23]. Likewise, we hope that Stench Blossom's use of negotiated interruptions, instead of immediate interruptions, will allow programmers to be more productive while programming.

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 [28]. 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 ↓ direction, and the smell that is least likely to be noticed without the use of a tool is shown

in the ↑ 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 [5]. 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" [22].

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.

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 [21], that display a single visualization of code smells. However, many existing smell detectors, especially ones that underline code that contains smells [6, 17, 27, 29], 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
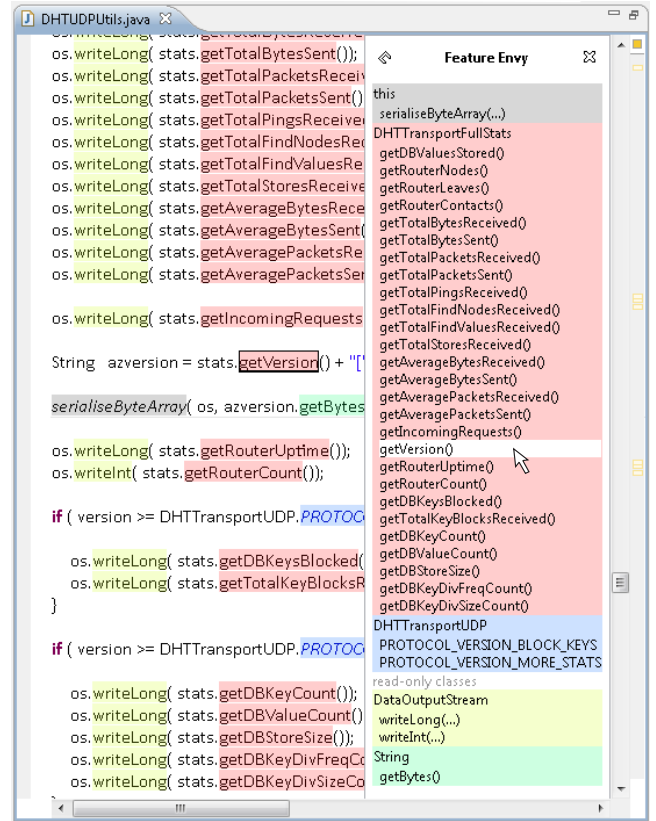

Figure 3: Explanation View for the smell FEATURE ENVY.

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 [24, 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" [25, 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 mov-

able 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 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 radius 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 [7], or Parnin and Görg's usage contexts [20].

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.

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 judgements based on smells, with and without Stench Blossom. To facilitate replication of this experiment, materials, including the codesets, experimenter's notebook, and results database, can be found at http://multiview.cs.pdx.edu/refactoring/experiments.

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 [3] implements only two smells (INSTANCEOF and TYPECAST); conducting a comparative experiment against just these two smells would produce quite limited results.

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.

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. 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. We chose codesets A and B to contain an approximately equal variety of smells. Each codeset contained 4 classes selected from the Vuze (http://vuze.org) and core Java libraries (http://openjdk.java.net/groups/core-libs).

The experiment started with a training phase, then had three parts in which we tested four hypotheses, as described below.

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 hearing 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 [12]. 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 [4].
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 [16]. 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 [4, 16], 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 [4]; 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.

4.2.1 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 [21], 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 [13]. 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 Judgements

Next, subjects made refactoring judgements 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 judgements 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 judgements 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 judgements. 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 judgements 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 [19], 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 judgement 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 [24, 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 [18].

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 tool than not using the tool. 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). 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 judgements when using the tool than when not using the tool. 10 out of 12 subjects said that the tool improved their confidence in refactoring judgements, and 11 out of 12 said that the tool helped them to make more informed judgements.

A feature that appeared help subjects make refactoring judgements 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 judgements. 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 judgements that were confident and informed.

The experiment confirmed **Hypothesis 4**, that the guidelines represent desirable design considerations for smell detectors. Overall, subjects rated our guidelines as important to the design of smell detectors. Table 3 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 3. Subjects tended to rank the 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

	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

How important is the characteristic to any smell detection tool?

Table 3: Results of post-experiment guideline questionnaire.

test), suggesting that programmers believe that our guidelines are generally important to usable smell detectors.

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 [4], 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 judgements 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. Further studies are needed to validate our results for different smells and with code more familiar to the subjects.

5. FUTURE WORK

We feel that future work on Stench Blossom could proceed in at least two directions. First, the tool itself can be improved in several ways. Second, the concepts used in the design of Stench Blossom may be beneficial in domains beyond smell detectors.

5.1 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 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 programmers do not agree with our ranking of the obviousness of smells, then the visualization could allow the programmer 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.[1] Using this information, the programmer would be made aware of the effect that her changes are having on the smelliness of the code.

5.2 Further Applications

We feel that the visualization technique and guidelines that we have presented in this paper are useful beyond the code smell tool that we have described. Specifically, we believe that our guidelines will be useful in any applications where users need to make judgements based on what the tool is telling them, and when the information to be presented is multi-faceted. These properties may hold for other tools used in software development and for a growing number of "recommender 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 [31]. 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).

An example from outside of software development is the user-interface of a grammar advisor, such as is often incorporated in word processors. The task is similar to informing the user about code smells: the display should be unobtrusive, show restraint and be non-distracting, so as not to interfere with the primary task of composition. Errors that arise from lack of parallelism in two different phrases or lack of agreement in the parts of speech should be shown relationally. The grammar advisor should tell me, first and foremost, about the text that I'm working on, and it should emphasize potential problems that are difficult to see with the naked eye. It should also help me estimate how much work it will take to fix some text: will the change be localized or will it ripple through the document? Moreover, the grammar advice should be available at all times, and should not only point out a problem but also explain why it exists.

---

[1]We thank Bill Pugh for this suggestion.

6. CONCLUSION

It is becoming increasingly recognized that programming is a design activity, and that it is the duty of every programmer who touches a code base to improve it. The alternative is "code rot," when the code base deteriorates as it evolves until it eventually becomes unmaintainable. Code smells can warn against such deterioration and encourage the redesign and refactoring necessary to "keep the code clean," but only when programmers are aware of code smells. In this paper, we have described a smell detector that uses an interactive ambient visualization to help make programmers aware of smells and make informed, confident refactoring judgements. Our evaluation suggests that such tools have a place in the software developer's toolkit. We hope that we have helped to clarify what that place is: not front and center stage, where they get in the way of the primary programming task, but in the background, always ready to offer advice on smells when requested, but keeping a low profile when the programmer is focused on other tasks.

7. ACKNOWLEDGMENTS

8. REFERENCES

[1] 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.

[2] 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.

[3] 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.

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

[5] 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.

[6] 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.

[7] 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.

[8] Andrew J. Ko, Htet Htet Aung, and Brad A. Myers. Eliciting design requirements for maintenance-oriented IDEs: a detailed study of corrective and perfective maintenance tasks. In Gruia-Catalin Roman, William G. Griswold, and Bashar Nuseibeh, editors, ICSE '05: Proceedings of the 27th

International Conference on Software Engineering, pages 126–135. ACM, 2005.

[9] 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.

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

[11] 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.

[12] 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.

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

[14] Daniel McFarlane. Comparison of four primary methods for coordinating the interruption of people in human-computer interaction. *Human-Computer Interaction*, 17(1):63–139, 2002.

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

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] Chris Parnin, Carsten Görg, and Ogechi Nnadi. A catalogue of lightweight visualizations to support code smell inspection. In Koschke et al. [9], pages 77–86.

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

[23] 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.

[24] 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.

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

[26] 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.

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

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

[29] Nikolaos Tsantalis, Theodoros Chaikalis, and Alexander Chatzigeorgiou. JDeodorant: Identification and removal of type-checking bad smells. In *European Conference on Software Maintenance and Reengineering*, pages 329–331. IEEE Computing Society, 2008.

[30] Richard Wettel and Michele Lanza. Visually localizing design problems with disharmony maps. In Koschke et al. [9], pages 155–164.

[31] 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.