A Catalogue of Lightweight Visualizations to Support Code Smell Inspection

Chris Parnin*
College of Computing
Georgia Institute of Technology
Atlanta, Georgia, U.S.A.

Carsten Görg†
College of Computing
Georgia Institute of Technology
Atlanta, Georgia, U.S.A.

Ogechi Nnadi‡
College of Computing
Georgia Institute of Technology
Atlanta, Georgia, U.S.A.

Abstract

Preserving the integrity of software systems is essential in ensuring future product success. Commonly, companies allocate only a limited budget toward perfective maintenance and instead pressure developers to focus on implementing new features. Traditional techniques, such as code inspection, consume many staff resources and attention from developers. Metrics automate the process of checking for problems but produce voluminous, imprecise, and incongruent results. An opportunity exists for visualization to assist where automated measures have failed; however, current software visualization techniques only handle the voluminous aspect of data but fail to address imprecise and incongruent aspects. In this paper, we describe several techniques for visualizing possible defects reported by automated inspection tools. We propose a catalogue of lightweight visualizations that assist reviewers in weeding out false positives. We implemented the visualizations in a tool called NOSEPRINTS and present a case study on several commercial systems and open source applications in which we examined the impact of our tool on the inspection process.

Keywords: Code inspection, lightweight visualization, code smells, refactoring.

1 Introduction

The software industry needs effective and practical tools to scaffold the process of maintaining quality software. To regulate cost and risks in software products, developers use a variety of techniques. One widely practiced technique, software inspection, systematically reviews the quality of source code. When performing an inspection, developers use intuition guided by past exposure to problematic structures and common design mistakes to uncover flaws. The use of checklists has been shown to be an effective approach for assisting developers in reviewing common errors [Mays 1990].

Unfortunately, design problems and source code quality are often considered to be less important than more pressing goals such as fixing bugs and adding features. As a result, few resources are devoted to inspections or addressing their findings.

Researchers and software practitioners find it desirable to replace the expensive human element with more automated methods of discovering software flaws. Object-oriented metrics [Lanza et al. 2005] offer a promising way of achieving this goal. However, metrics are not as forgiving or introspective as their human counterparts. The defects discovered by analysis have high false positives [Flanagan et al. 2002; Rutar et al. 2004; Kim and Ernst 2007] and can be mis-aligned with human judgment [Mäntylä et al. 2004].

Metrics have reduced problems with manual inspection but they also have created a new problem: inspecting the output of thousands of defect warnings. In experiments with a bug finding tool, `ESC/Java` [Flanagan et al. 2002], users complained about "excessive warnings about non-bugs" to the extent that severely impacted adoption. Rutar and colleagues [2004] noted in a comparison of different bug finding tools that the results contained "too many warnings to be easily useful by themselves". Kim and Ernst [2007] revealed in an analysis of several software archives that less than 10% of warnings were addressed and warning prioritization by tools offered little predictive power. Overall, these observations were consistent with our experience with professional developers at a company: frustration with warning systems generating too many results.

In this paper, we focus on code smells [Fowler et al. 2001], a set of symptoms indicative (but not confirmatory) of the presence of certain design problems. For each code smell, we design a simple light-weight visualization that can be used to complement an instance of a code smell warning found by an analysis tool. The visualization provides additional evidence that can be considered by a reviewer to determine the relevance, severity and accuracy of a warning.

The main benefit of our approach is the compactness and transportability of the visualizations: for example, an existing warning report can improve confidence by simply embedding one of our visualizations alongside each warning. The visualizations can be used in automated notification reports to provide summaries of recent warnings found during continuous testing. Finally, code reviewers can use our visualizations with defect analysis tools to assist in screening defect warnings.

The main contributions of this paper are:

1. A novel approach for visually inspecting defect warnings that allows developers quickly filter out false positives;

2. A catalogue of lightweight and coherent visualizations for code smells;

3. A tool for detecting and visualizing code smells that was tested on several commercial systems and open source applications.

In the next section, we describe background material and work related to code smells. In Section 3, we present results of a design study to inform our techniques. In Section 4, we introduce a catalogue of lightweight visualizations and we discuss a case study describing the experiences in using our tool in Section 5. Finally, in Section 6, we draw conclusions and discuss further research directions.

*e-mail: chris.parnin@gatech.edu
†e-mail: goerg@cc.gatech.edu
‡e-mail: ogechi.nnadi@gatech.edu

SOFTVIS 2008, Herrsching am Ammersee, Germany, September 16–17, 2008.

2 Related Work

Several tools have been proposed to assist developers with inspecting the quality of source code. Early examples such as LINT [Johnson 1978], detected static errors common in code but focused only on low-level implementation flaws. In response, researchers attempted to find more design-oriented measurements. Dromey [1995] proposed a set of quality metrics based on the structural form of code to allow classification and automation of quality defects. Riel [1996] defined a set of object-oriented guidelines for monitoring the quality of code design, and Fowler and colleagues [2001] introduced code smells.

Code smells are structural indications of larger design problems lurking in the code. Experienced developers may notice the smells emerge as they are making modifications to source code and find more than undue resistance to their efforts. Other smells originate from the code structure: a large class, for example, may contain several concepts that should be abstracted and reused in other places. Other times, faulty composition and coordination of classes and features suggest reconsiderations in assigning responsibilities.

Mäntylä *et al.* [2003] break down the different code smells into general classes of problems:

Bloaters: The code structure begets system entropy;

OO-Abusers: The code violates object-oriented principles;

Change Preventers: The code structure prevents changes;

Dispensables: The code structure contributes no value;

Couplers: The communication patterns of classes are problematic.

Several inspection systems have addressed the detection aspect by automatically detecting design flaws [Crespo et al. 2005; Slinger 2005] or by supporting querying for code smells [Tourwé and Mens 2003]. Additionally, many systems also use visualization to assist presenting metric data. For example, Demeyer *et. al.* [1999] used CODECRAWLER to visualize code quality metrics.

Generic metric visualizations offer some assistance, but neither scaffold the inspection process nor offer a checklist approach letting the reviewer check for a specific type of flaw. The reviewer is expected to see a problem in the metric representation rather than weigh evidence for or against a specific design flaw.

Keeping the *human in the loop* follows Fowler's advice of using human intuition for detecting code smells [Fowler et al. 2001]. Techniques that allow exploration and queries, such as the REFACTORING BROWSER [Tourwé and Mens 2003], incorporate this advice. JCOSMO [Emden and Moonen 2002], a system to visualize code smells in source code, displays the code structure as a graph and maps code smells onto the attributes of that graph. This approach can be problematic for several reasons. The visualization is built assuming that code smells are concentrated in a particular region of the code and that metrics will point reviewers there. This assumption does not always hold; many code smells require understanding the relationships between many interacting components and thus are spread throughout the program. These relationships cannot be represented by a simple mapping between code structure and color. In addition, the reviewer must explore this structure without any semantic guidance, smell-specific scaffolding, or checklists.

Other researchers designed visualizations for specific code smells. Simon *et. al.* [2001] tackle coupling smells with a 3D visualization of feature associations. This approach is sound; however, it has trouble scaling to more than a few classes. Micro-prints [Ducasse et al. 2005] are designed to represent elements of code statements as pixels. In an evolution matrix [Lanza 2001], class change patterns are visualized to reveal interesting trends in how classes change over time. Both visualizations reveal interesting patterns but are very specialized in what problems they solve.

In summary, although attempts have been made to visualize code smells, they handle few smells, fail to address imprecision in the metrics, and do not design the visualizations with software inspection in mind. Furthermore, these approaches also fail to scale down. They are intended for a system-wide view instead of also being capable of illustrating one design problem. In our work, we handle more code smells from a variety of categories and engineer our visualizations to be suitable for use in software inspections.

3 Design Study

In this section, we describe different facets of software inspection observed in practice and the different challenges they pose. From these challenges, we compose a set of requirements that a software inspection tool should satisfy. In our study, we measured the nature of warnings reported by an automated analysis tool in comparison with our observed peer reviews.

3.1 Software Inspection Study

The following observations are drawn from the collective experiences of the authors and four other employees participating in over 40 peer reviews when working at a software company.

3.1.1 Code Peer Review

A common code inspection technique is to perform a *desk-check*, where a peer examines code on their own time. After a desk-check, findings can be discussed in person or at a peer review meeting (in contrast to a more strenuous *structured walk-through*).

During our study, desk-checks were routinely scheduled as part of the development process. Many reviews were scheduled after design phases focusing on revealing architectural issues. Reviews were also scheduled after the completion of a module or before a delivery. Finally, desk-checks were scheduled by supervisors to monitor less experienced employees or to investigate an abnormally faulty module. Manually monitoring was difficult to manage, often causing mistakes not to be caught until much later in development.

The time allocated for desk-checks was constrained by the project budget and employee availability. Experienced employees were sought for reviews, but had limited time to devote and added additional expense onto a project budget. These factors resulted in reviewers typically spending between one and four hours and rarely one work day. Review meeting sessions were limited to two hours in length, with no review meetings lasting more than three sessions.

The outcome of peer reviews varied. Reviews addressing correctness of algorithms and implementations produced effective results. Design reviews suffered from limited time to comprehend large designs and lack of a shared vocabulary to discuss design flaws. The number of accepted findings ranged from 0 to 200 but was typically under 50.

3.1.2 Continuous Testing

Continuous testing is the process of automatically creating software builds and running tests in a frequent manner. This process is designed to bring immediate attention to integration problems, bugs, and quality problems.

During our study, several software teams made use of continuous testing. One problem we observed related to the notification of vi-

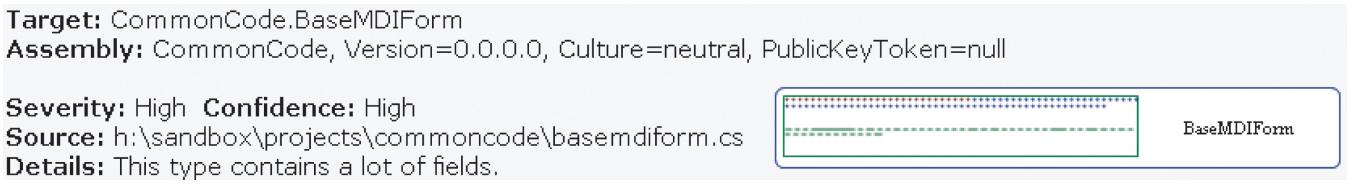


Figure 1: *Warnings can be made more informative by embedding mini-visualizations.*

olation of quality metrics. Warnings were being issued at a low volume but high frequency. The project managers were receiving daily warnings but did not know how to trust the veracity of the warnings or how to precede with delegation. In many cases, the notification was eventually disabled.

We draw the following implications for design from our study:

1. Reviewers typically have less than four hours to review.
2. Monitoring is a common activity.
3. Reviewing design quality is difficult to perform manually in a short period of time.

3.2 Automatic Design Warnings Experiment

To understand how automatic tools worked in comparison with manual peer reviews, we conducted a study on a commercial system with 140 KLOC (668 classes) using an open-source tool called GENDARME.[1]

The tool used 109 rules and reported 5038 defects in an html report. 1467 locations were affected with defects. The html report contained 21366 non-blank lines of text. This is roughly half the 47293 non-blank lines of text in the epic novel "War and Peace".[2] The largest category with 2031 defects belonged to duplicated code rule. Other defects included 157 large classes, and 88 long methods. This number of defect warnings is in contrast to the 50 findings typically accepted in our observations of peer reviews.

When applying existing visualization techniques to these findings, we had several observations. In one example, a treemap was useful in viewing the distribution of the 157 detected large classes across the code base (see Figure 2). But it was difficult to assess the severity or accuracy of warnings. The tool uses multiple criteria to detect defects, such as the number of member variables in a class or the size of a method. However, appropriate values for these attributes vary depending on several factors, such as the type of a class (domain class versus UI class). This important information was not observable in the treemap. Finally, we noted that the visualization did not assist in tracking visited problems and managing fixes.

4 Catalogue of Code Smell Views

In this section, we present a catalogue of code smell visualizations. We divide the smells into four different categories: statement, class, method, and collaboration code smells. In each category, we first discuss the common insights and visualization elements that this category of code smells shares. Then we describe the code smell, present its abstract representation, and illustrate it using a set of examples drawn from actual code produced by industry.

[1] http://www.mono-project.com/Gendarme

[2] http://www.jus.uio.no/sisu/war_and_peace.leo_tolstoy/plain.txt

Figure 2: *Traditional visualizations excel in showing distribution of metrics across a code base. But when false positives can be as high as 90%, users need more detail to understand the warnings.*

4.1 Design

Considering the the nature of the warnings, we sought to design visualizations that could handle large number of warnings while maintaining the necessary level of information fidelity needed to assess a warning. The design criteria used in creating the code smell visualizations focus on information fidelity, scalability, and ability to discuss trade-offs. This decision was further motivated by the fact that many software developers may not be familiar with code smell terminology or reach consensus on the presence of a smell.

We also considered multiple usage scenarios when designing our visualizations. In one scenario, a report generated by an defect analysis tool such as GENDARME can include a visualization alongside the warning to provide more context (see Figure 1). In another scenario, project managers receiving daily notifications of warnings could use the visualizations to screen the warnings (see Figure 3). Finally, code reviewers can use the visualizations as part of a code inspection tool. We examine this aspect later in our case study in Section 5.

The motivation for using a simple set of visual elements was to accommodate the diverse nature of warnings types. The basic visual symbols used in our design are shown in Figure 4.

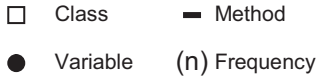


Figure 4: *These basic symbols are the building blocks for our visualizations.*

4.2 Statement Code Smells

Certain design problems are only demonstrated when examining the code statements inside method bodies. Traditional code metrics

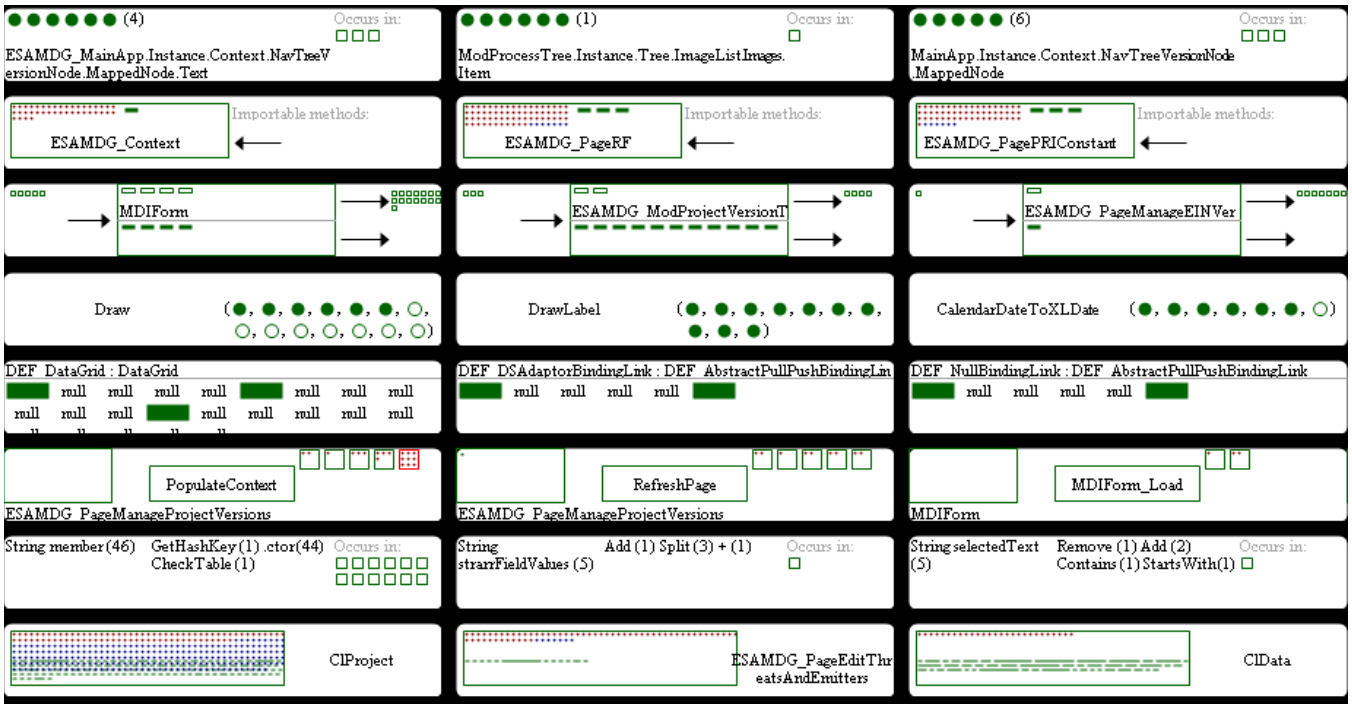


Figure 3: *A simple health screen can allow project managers to quickly and frequently gauge the health of their projects.*

such as cyclomatic complexity do reveal potential problems with overly complex code; however, many more design problems exist besides statement complexity.

Design problems are often not identifiable by viewing a design element in isolation but rather need to be examined in context. For example, sometimes keeping data in a seemingly innocuous location inadvertently becomes a problem when data requests must navigate a convoluted path to acquire that data. This problematic path is often referred to as a *Message Chain*.

Two important facts are relevant in diagnosing design problems symptomatic of statement code smells: the *spread* of the code smell throughout the program, and the *frequency* of the same code smell instance. The spread of the code smell allows a reviewer to consider how widespread and how difficult a problem may be to fix.

In our visualizations, we denote spread by showing how many classes a smell resides in using the visual element shown in Figure 5. The number of classes give a better measure of spread than the methods because with only methods, the reviewer cannot distinguish smells occurring in many methods in the same class versus a smell occurring in several methods across many classes. Spread is displayed graphically because we want to support code reviewers incorporating this criterion when visually scanning the code smells. Code reviewers are more likely to notice the graphical representation due to faster perceptual response to shapes than to text. Frequency of occurrence is considered secondary and thus is denoted by a numerical value.

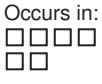
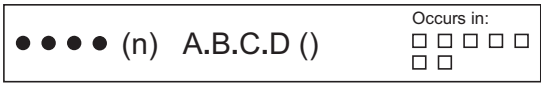


Figure 5: *The spread shows the classes where the code smell occurs.*

Finally, when examining a specific instance of a statement code smell, the most important aspect is the *text* of the code statement. The text contains essential semantic clues for the reviewer when weighing whether a metric is a false alarm or a serious problem.

4.2.1 Message Chain



Smell: A message chain is a statement that contains a long sequence of method invocations or instance variable accesses. Long message chains are problematic because they expose unnecessary dependencies and may introduce data access bugs.

Context: The reviewing task is discover to problematic accesses to data and to consider alternative mechanisms for storing and retrieving data.

View: All message chains found in a project are grouped according to their constituent objects in a list. The dots on the left indicate the length of the chain (number of involved variables or methods called), the number in parentheses shows how often the chain occurs, and the dot-separated characters represent the chain itself. The spread is displayed on the right.

Review: In Figure 6, the first two message chains should be investigated. Both appear to be accessing potentially volatile information: the chains are accessing data through user interface elements that may be destroyed or become inactive. This may lead to crashes or deletion on incorrect data.

The third message chain is long. Although at first glance this may be alarming, this type of statement is typical for XML processing. Because the occurrences are relatively low and isolated, this smell is a low priority problem unless its frequency or spread increases.

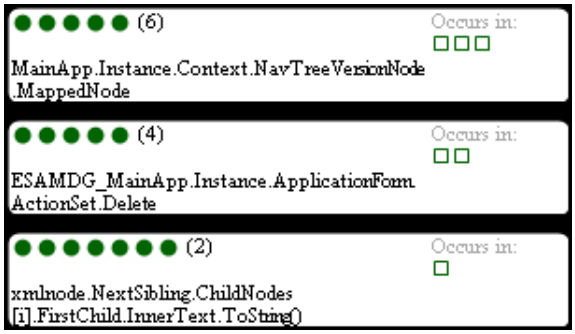
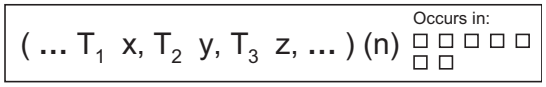


Figure 6: *The third message chain is problematic but low-priority.*

4.2.2 Data Clumps



Smell: A data clump is a group of variables that is often passed to methods together. For example, a graphics program could have methods that commonly use parameters `(int red, int green, int blue)` to represent a color.

Context: The reviewing task is to find similarly named groups of variables in parameter lists and to replace them with one parameter variable. Thus, in the example above, we would find all methods that use `int red, int green, int blue` as parameters and replace them with a variable of type `Color`.

View: Each data clump is listed and shows the types and names of the data clump members, the number of clump occurrences, and the spread.

Review: We did not find occurrences of this smell in our case study. Figure 7 illustrates the concept.

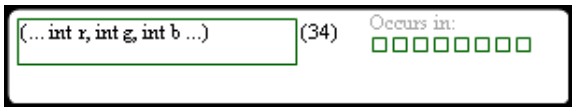
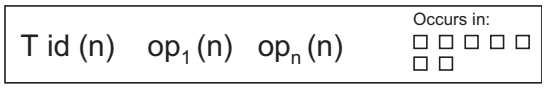


Figure 7: *Color object should be created: in 8 classes, 34 methods pass around r,g,b.*

4.2.3 Primitive Obsession



Smell: Sometimes a variable with a primitive type is used to encode data that would be better stored in the fields of a class. For example, a string could be used to represent an amount of money, such as "$500"; however, it would be better to store it as an object with currency and amount variables.

Context: The reviewing task is to find instances of a primitive on which the same extraction operations are performed repeatedly.

View: The primitive variable is shown with the frequent operations performed on the variable. The spread of occurrences shows how many classes are concerned with the data in that variable (potential aspect?).

Review: In Figure 8, the first two primitives are acceptable. The first primitive holds a line of text obtained via parsing an input file, so naturally many operations must be performed on its content. The second primitive holds a file name and limits its operations to file processing – nothing wrong. The third primitive appears to be involved with logging and recording notes. The message is combined with other data but it is not unpacked. This smell should be investigated to see if a common message class would help.

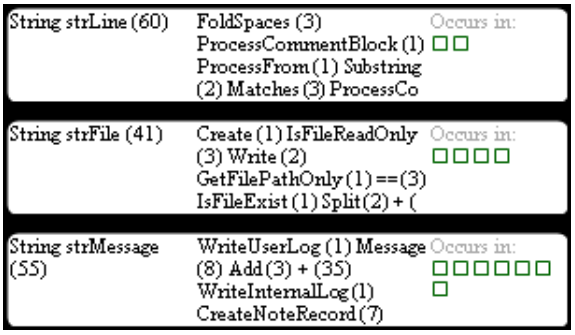


Figure 8: *Should a message class be used to better handle messaging?*

4.3 Class Code Smells

Class code smells typically involve mishandled domain concepts. The developers have created code that poorly represents a concept or contains too many concepts. Examining the methods and fields of the class offers insight into possible problems. In Figure 9, we show how we represent classes and its methods and field variables. However, detecting problems is not as clear-cut as putting limits on size since there are always exceptions. For example, user-interface classes are more likely to contain many field variables because those are often auto-generated for each user-interface element such as a `Label` or a `TextBox`. To distinguish these two types of fields we color user-interface derived fields blue and other fields red.

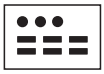


Figure 9: *Class symbols can contain other symbols. This class has three field variables and six methods.*

4.3.1 Data Class

Smell: A data class stores many public instance variables but does not provide much functionality. Methods from other classes could be moved into this class to make it more useful.

Context: The reviewing task is to know the number of instance variables and methods in a class to decide whether the class is performing enough work (getters and setters—methods that only return or set the value of an instance variable—do not count as useful methods). We use the following heuristic to find methods that would be good candidates for inclusion into the class: the methods take the class as a parameter and do not reference any instance variables of the class in which they are contained.

View: All project classes are listed and show the number of variables and methods for each class and a list of candidate methods for inclusion into the class.

Review: In Figure 10, the first class should no doubt be checked. Even if it is not a data class, the large number of variables hint at some design problem. The second class exhibits signs of a classic data class. There are many instance variables, but only one method. In addition, there are many external methods processing data from the `Marker` class that should probably be imported into the class. The third class has more instance variables than the second class, but it also has more methods; overall, the class appears relatively healthy.

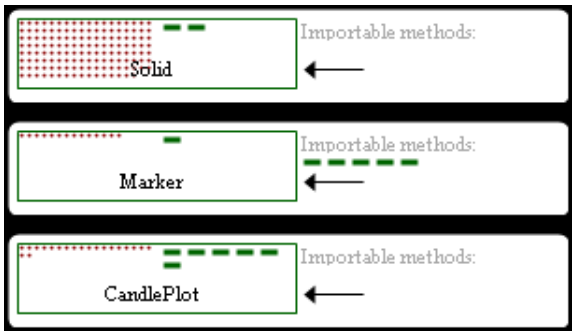


Figure 10: *The Marker class should start acting like a class and import some methods.*

4.3.2 Large Class

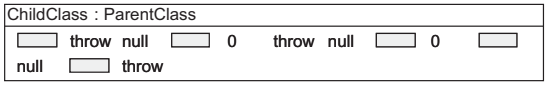
Smell: A class that is too large to work with.

Context: The reviewing task is to determine if a class is too large. Metrics for determining the size of a class include the number of instance variables, the number of methods, and the method size distribution.

View: The classes in the project are listed in decreasing order of lines-of-code. The dots display the instance variables and the bars displays each method's length allowing the reviewer to notice anomalies with the method size distribution.

Review: In Figure 11, the first class has a few field variables and several long methods but nothing really stands out. The second class is a classic example of a *kitchen-sink class*, many utility methods are thrown into a single class instead of the class modeling a specific concept. The third class is a regular class. The class only has a few real instance variables, the blue fields are user-interface fields such as a `TextBox`.

4.3.3 Refused Bequest



Smell: A class manifests a refused bequest if it inherits methods but does not use them. Fowler *et al.* [2001] consider this a weak smell. In our definition, we instead focus on cases where a class assumes an interface, but does not properly support the interface by either throwing a not implemented exception, returning Null, or returning a constant.

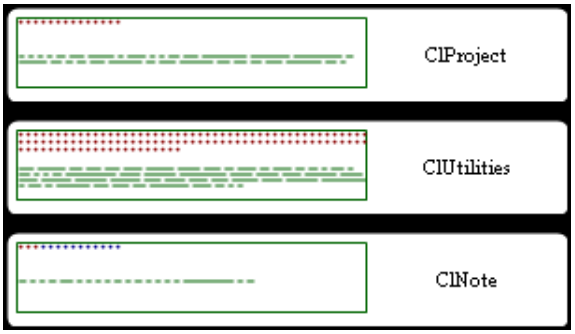


Figure 11: *Too much clutter has built up in a kitchen-sink class (second class).*

Context: The reviewing task is to determine the ratio of implemented to "stubbed out" interface methods in order to judge the severity of a smell.

View: For each class all implemented interface methods are listed. "null", "0", and "throw" indicate a refused bequest wheres a green bar indicates regular behavior. This kind of highlighting allows the reviewer to discern easily whether a large number of methods have been stubbed out.

Review: The first class in Figure 12 is a well-behaved child class. The second class in contrast wants to be considered a `Project` but it does not actually implement the required methods. This behavior suggests some conceptual abuse of the project concept is occurring. The third class is on the fence, further manual inspection is needed.

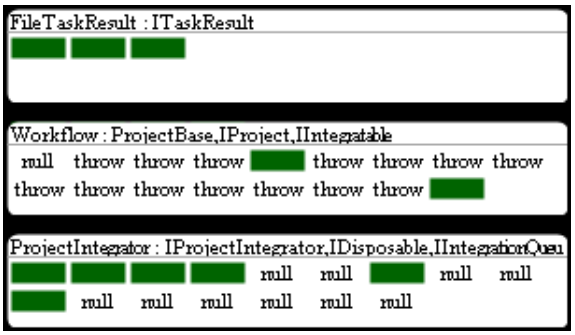
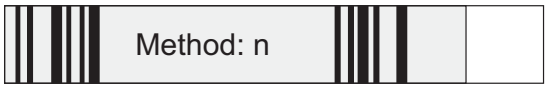


Figure 12: *There are some conceptual modeling problems when classes start borrowing interfaces (second class).*

4.4 Method Code Smells

Method-level code smells are relatively easy to understand. Classic code metrics and line-based visualizations have long been used to representing them. Unfortunately, line-based visualizations take up too much vertical screen real estate that makes it virtually impossible to display the method name without a detail-on-demand query or sideways text. We believe that the ability to see the method name outweighs the benefits of the mental model offered by line-based visualizations. Thus, we turn the lines sideways.

4.4.1 Long Method



Smell: A method that is too long to work with. The method is often difficult to understand and may contain duplicate code.

Context: The reviewing task is to determine the number of lines of code in the method and to understand its cyclomatic complexity.

View: The length of a method is mapped to a gray bar. For every control flow statement, conditional statement, and comment in the method, a line is drawn at the corresponding position along the bar's length to give a sense of the structure of the method. Each line is marked with a color to indicate its type: light blue denotes conditions, blue denotes iterations, green denotes comments, and gray other statements. The number of lines of code are also shown numerically after the colon.

Review: In Figure 13, the first method is very long; however, it is an auto-generated method for creating user interface elements. Normally this would be fine, but there might be too many user interface elements lumped into this one class. The second method is of moderate length and mild complexity. In the third method, neither the length, nor the complexity is particularly alarming; the fact that this code is for parsing longitude and latitude values has such a complex implementation is alarming. The code is likely duplicating the logic for handling longitude and latitude, and might be better if replaced with regular expressions to extract the values instead.

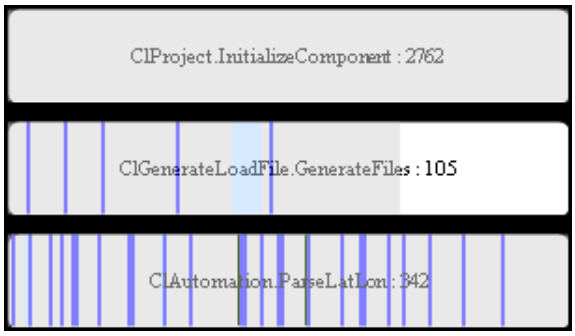
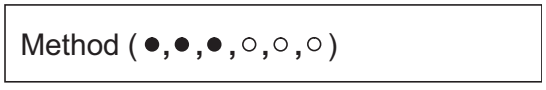


Figure 13: *Why is parsing longitude and latitude values (third method) so complex? Use regular expressions?*

4.4.2 Long Parameter List



Smell: A parameter list is too long to work with.

Context: The reviewing task is to determine if the parameter list is justified.

View: Parameter lists are represented by a row of circles, one for each parameter in the largest overload for that method. The number of filled circles corresponds to the number of parameters in the shortest overload of that method. All parameter lists are displayed and sorted in descending order of length.

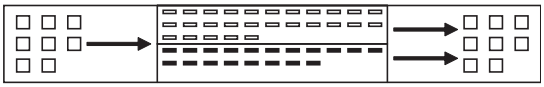
Review: In Figure 14, the first method contains an extraordinarily long list of parameters. Furthermore, the data appears to be unpacked into arrays. Something fishy is going on. The second method is moderately long; however, a simpler single parameter version is available, the longer version is most likely providing customizable options. The third method has a typical method signature.

Figure 14: *A method with a long parameter list has a more convenient overload (second method).*

4.5 Collaboration Code Smells

Collaboration code smells occur when classes interact in a strange fashion. The root cause is usually a mis-allocation of responsibility. Understanding the situation usually requires understanding a class and its interaction with clients and services.

4.5.1 Middle Man



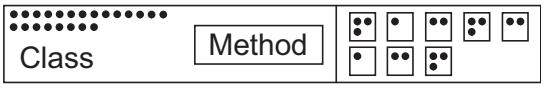
Smell: A middle man is a class that delegates most of its work to other classes. Middle men can often be removed and their functionality placed in the classes that they were mediating.

Context: The reviewing task is to determine the contribution of the middle man's functionality in comparison with its delegation to external classes. Knowing about clients and services gives better context to the usefulness of the middle man.

View: The middle man is the class in the middle. The methods on the bottom of the class are regular methods. The transparent methods on the top however, are not contributing anything and instead are just making requests to other classes. Clients are the classes on the left and servers are classes on the right.

Review: In Figure 15, the first class is not a middle man; however, the class seems to have a large number of methods that access a large number of services. An investigation should examine if smaller classes should be extracted from this class. The second and third classes both provide a type-safe collection class. However, in each case only one single client is using the class. Thus, these two classes may be eliminated.

4.5.2 Feature Envy



Smell: A method that makes many calls to other classes rather than calling methods within its own class is considered to be envious of other classes features.

Figure 15: *Is the developer being over zealous in creating a type-safe collection class (second and third class)? The collection is only used by one class.*

Context: The reviewing task is to determine for a given method the number of referenced members from other classes in comparison to referenced members in its own class.

View: On the left side, referenced members of the method's class are shown. On the right side, referenced members in external classes are shown. In our implementation, the external members are fields and getter/setter methods. Because of the limited space for showing members, a class is highlighted in red if its features exceed the capacity of the class representation.

Review: In Figure 16, the first method is not using any features of its class while accessing the features of other classes. The second method is using features internal to the class as well as several other classes. After examining these two methods, it is curious that both methods are processing tree information outside of a tree class. The third method looks normal.

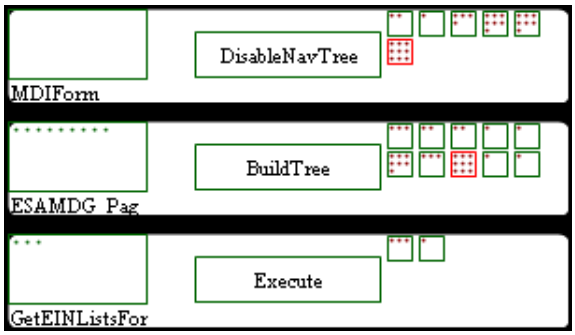


Figure 16: *Why are navigation trees being operated on outside of a tree class?*

5 Application

We implemented a suite of tools for supporting software inspection. We developed a byte-code scanner and source code scanner for detecting code smells, and a Visual Studio plug-in called NOSEPRINTS for displaying and reviewing code smells. We performed a case study with three industrial projects and two open source projects. In the study we examined the scalability of our approach on real world projects and the feasibility of using our tool in a design peer review.

The examples shown in Section 4 are real problems drawn from these projects. Many of these problems were serious enough to be addressed and did contain actual bugs caused by design flaws.

5.1 NosePrints

NOSEPRINTS is a Visual Studio plug-in for inspecting code smells. To detect the code smells, we use a combination of source code analysis and byte-code analysis [Parnin and Görg 2008] to obtain both type-rich analysis and the syntax information necessary in calculating metrics for code smells.

NOSEPRINTS is easy to use and to embed into an existing work flow: the reviewer first loads the analysis results and then follows the checklist of code smells to study each code smell one by one.

For example, if a reviewer wants to inspect code for any *Feature Envy* smells, then the tool provides the visualization shown in Figure 17. The developer then visually scans the results for something that catches their attention.

The reviewer can mark a warning to save for later review or choose to ignore it entirely. Later, the reviewer can access the code smells saved for review and take a more in-depth look by clicking on the code smell to see more details and links for inspecting the source code. If manual inspection is immediately desired, then clicking on the code smell displays links to relevant code sections in a code editor window in Visual Studio.

5.2 Large Projects

We examined how the code smells views scale for large projects produced industrially: one project had 80 KLOC (111 classes), the other 140 KLOC (668 classes).

We were concerned with how well the code smells views worked with code from real programs. Practical concerns included the questions: Would names fit within the view? How many occurrences of a potential smell would a reviewer have to examine? How many views fit on a screen? Does the representation of classes, methods, and variables fit within the allocated space? Was the right scope chosen for framing the smell?

Names were surprisingly long. In practice, the length of method names and classes makes it impractical to display for anything other than essential elements. In general, by wrapping names that were too long, we were able to display most names that we encountered. A frequent reason for longer names are common prefixes of classes. Trimming prefixes could save more space.

The number of occurrences a reviewer would have to examine is largely dependent on the unit of code structure that a code smell could exist in. For example, to find large classes, in the worse case the number of views the reviewer inspects is equal to the number of classes in the program. However, several factors contribute to a more moderate number of examinations needed by the reviewer. First, sorting quickly pushes irrelevant cases to the bottom. In the example of *Large Class*, the distribution of class size is such that there are fewer medium-to-large classes than small classes. Second, the nature of a code smell eliminates many candidates. For example, in *Refused Bequest* only classes that inherit or implement an interface are eligible.

In our implementation, using a size of 300 by 50 pixels for a view allowed 48 views to be displayed on a 1268 by 1048 resolution display. This allowed 43% of the 80 KLOC program's classes and 7% of the 140 KLOC program's classes to be examined per screen. With large wall-mounted monitors or projection screens, many more views can be accommodated and examined by multiple reviewers.

An interesting issue was discovering how well the representations responded to overflow when the default space for displaying data

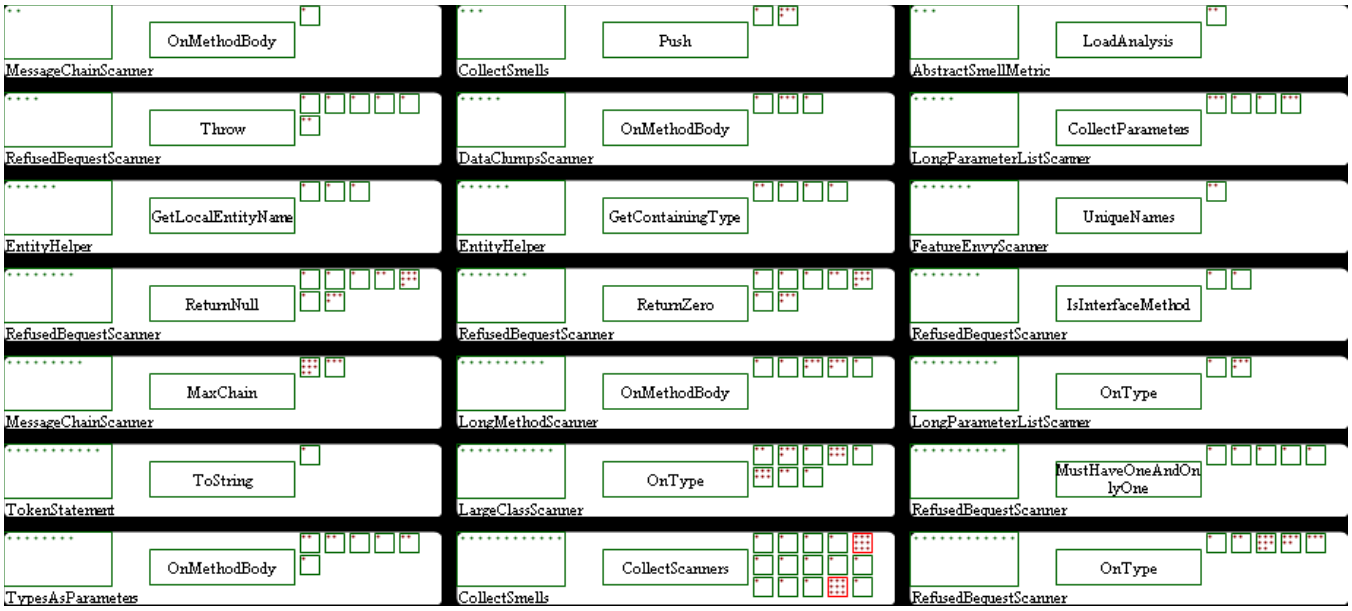


Figure 17: *A list of methods being screened for Feature Envy code smells.*

was exceeded. In general, the visualizations allows representations such as variables to overflow into other parts of the view and still make sense. Furthermore, because of the design of the views, it is rare for other representations to exceed their space. For example, in the *Data Class* view (see Figure 10), when a class has more than about 150 data members, then the data members overflows into the space for methods. The methods are still visible, and then the capacity can handle over 300 data members. On the other hand, it is not much of a concern for methods to overflow their bounds because the class would have far too many methods to be considered a data class.

In our last question, some initial choices in scope of framing smells became apparent. For example, in the *Long Method* view two problems were observed. First, the wrong level of abstraction was chosen: by default, it showed classes and their long methods. The resulting views gave too high of an overview, making selecting the longest method difficult. Second, complexity information was not visible – the reviewer could not easily distinguish test cases and auto-generated code from real problems. We incorporated these insights in our current implementation.

Finally, we found further improvements that could be made to the underlying metric analysis. Most cases involved taking more advantage of the type and attribute information to better understand or filter out certain cases. In one example, highlighting field variables that are constant would better identify classes that are declaring many values versus storing properties. In another example, identifying which types associated with *Primitive Obsession* variables would allow reviewers to identify where the value was originating from (file, object, UI). Lastly, highlighting common fragments of message chains would improve measuring severity and identifying root causes.

5.3 Peer Review

One of the authors was invited to perform a peer review on a recently completed project. The project was relatively small, containing 7000 LOC (39 classes). The author examined the project along with five other reviewers. Each reviewer only had a limited amount of time (two hours) they could charge to the project. The author used NOSEPRINTS to structure the inspection process, whereas the other reviewers manually inspected the code.

Using the tool, a total of 22 findings were reported by the author using NOSEPRINTS. All findings were accepted at the meeting to be addressed. One remarked that this code review was one of the most useful code reviews they had experienced. The other five reviewers reported from zero to two problems. This does not prove that the tool is better than manual inspection, but demonstrates evidence that the tool can be used within the constraints of a typical peer review.

6 Conclusion and Future Work

In this paper, we present a catalogue of code smell visualizations to support code inspections. As more sophisticated metrics are developed, new techniques are needed for handling the results of the metric analysis. Furthermore, it is not sufficient to directly visualize the code structure and expect insight to emerge from that picture. Visualizations need to be crafted to piece together a coherent story specific to underlying problems that the metrics were measuring. Many software engineers are not familiar with the vocabulary and insights of the more sophisticated metrics. The role metric visualization systems have is in part educational and part communicative.

In some sense, our visualizations depart from traditional software visualization approaches: instead of using an exploratory interface, we advocate self-contained views that display contextual information without requiring interaction. We find our approach scalable enough for use in typical real-world systems and useful in performing a peer review of a medium-sized project.

Our catalogue does not cover all the code smells described by Fowler *et al.* [2001]. NOSEPRINTS does cover *Temporary Fields*, *Inappropriate Intimacy*, and *Parallel Inheritance Hierarchy*—we did not include them in our catalogue due to space constraints. Other smells are either overly broad or require a more specialized detection method. For example, the smell *Duplicate Code* is a symptom of a deeper problem; after examining a possible visualiza-

tion, we think a more fine-grained classification is needed to break down the task of detecting duplicated code. The smells *Divergent Change* and *Shotgun Surgery* are best detected by examining several revisions of the source code because these smells are only detectable after it has proven difficult to modify code. Finally, the smells *Dead Code* and *Lazy Class* are different extremes of *Speculative Generality* which in itself is subjective in terms of how code may be used in the future.

We are exploring how to better integrate our visualization into the software development process. We are currently working on integrating our views with CRUISECONTROL, a continuous integration system, so that users can be notified of possible quality problems. We experimented with a technique [Parnin and Görg 2006b] for applying *usage contexts* [Parnin and Görg 2006a] taken from developers' interactions with an integrated development environment to favor the display of recently visited or edited methods when returning code search queries. Finally, we have explored ways of getting overview with summaries and stacking views into piles (see Figure 18) and improving navigation. With this additional support, developers can have a better overview and also focus on inspecting more relevant code.

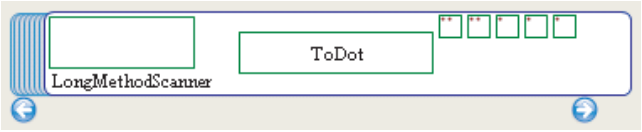


Figure 18: *Stacking allows warning instances to be grouped by different attributes such as under same class.*

Acknowledgements

Spencer Rugaber and the anonymous reviewers gave helpful comments on an earlier version of this paper.

References

CRESPO, Y., LÓPEZ, C., MARTICORENA, R., AND MANSO, E. 2005. Language independent metrics support towards refactoring inference. In *QAOOSE '05: Proceedings of the Ninth ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, 18–29.

DEMEYER, S., DUCASSE, S., AND LANZA, M. 1999. A hybrid reverse engineering approach combining metrics and program visualization. In *WCRE '99: Proceedings of the Sixth Working Conference on Reverse Engineering*, 175–186.

DROMEY, R. G. 1995. A model for software product quality. *IEEE Transactions on Software Engineering 21*, 2, 146–162.

DUCASSE, S., LANZA, M., AND ROBBES, R. 2005. Multi-level method understanding using microprints. In *VISSOFT '05: Proceedings of the 3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*, 33–39.

EMDEN, E. V., AND MOONEN, L. 2002. Java quality assurance by detecting code smells. In *WCRE '02: Proceedings of the Ninth Working Conference on Reverse Engineering*, 97–106.

FLANAGAN, C., LEINO, K. R. M., LILLIBRIDGE, M., NELSON, G., SAXE, J. B., AND STATA, R. 2002. Extended static checking for Java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, 234–245.

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

JOHNSON, S. 1978. Lint, a C program checker. Tech. rep., Bell Laboratories, Computer Science.

KIM, S., AND ERNST, M. D. 2007. Which warnings should I fix first? In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 45–54.

LANZA, M., MARINESCU, R., AND DUCASSE, S. 2005. *Object-Oriented Metrics in Practice*. Springer-Verlag New York, Inc.

LANZA, M. 2001. The evolution matrix: recovering software evolution using software visualization techniques. In *IWPSE '01: Proceedings of the 4th International Workshop on Principles of Software Evolution*, 37–42.

MÄNTYLÄ, M., VANHANEN, J., AND LASSENIUS, C. 2003. A taxonomy and an initial empirical study of bad smells in code. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, 381–384.

MÄNTYLÄ, M. V., VANHANEN, J., AND LASSENIUS, C. 2004. Bad smells - humans as code critics. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, 399–408.

MAYS, R. G. 1990. Applications of defect prevention in software development. *IEEE Journal on Selected Areas in Communications 8*, 2, 164–168.

PARNIN, C., AND GÖRG, C. 2006. Building usage contexts during program comprehension. In *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension*, 13–22.

PARNIN, C., AND GÖRG, C. 2006. Lightweight visualizations for inspecting code smells. In *SoftVis '06: Proceedings of the ACM Symposium on Software Visualization*, 171–172.

PARNIN, C., AND GÖRG, C. 2008. Improving change descriptions with change contexts. In *MSR '08: Proceedings of 5th Working Conference on Mining Software Repositories*, 51–60.

RIEL, A. J. 1996. *Object-Oriented Design Heuristics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

RUTAR, N., ALMAZAN, C. B., AND FOSTER, J. S. 2004. A comparison of bug finding tools for Java. In *ISSRE '04: Proceedings of the 15th International Symposium on Software Reliability Engineering*, 245–256.

SIMON, F., STEINBRÜCKNER, F., AND LEWERENTZ, C. 2001. Metrics based refactoring. In *CSMR '01: Proceedings of the Fifth European Conference on Software Maintenance and Reengineering*, 30–38.

SLINGER, S. 2005. *Code Smell Detection in Eclipse*. Master's thesis, Department of Software Technology, Delft University of Technology, Netherlands.

TOURWÉ, T., AND MENS, T. 2003. Identifying refactoring opportunities using logic meta programming. In *CSMR '03: Proceedings of the Seventh European Conference on Software Maintenance and Reengineering*, 91–100.