PyReflect: An Implementation and Comparison of Java Static Code Smell Detectors

[Final Report]

Chris Buonocore
University of California, Los Angeles
cbuonocore@ucla.edu

Meghana Ginjpalli
University of California, Los Angeles
mginjpalli@ucla.edu

Han Wang
University of California, Los Angeles
hannahoo@ucla.edu

ABSTRACT

According to Martin Fowler, one of the founding leaders in code refactoring, a code smell is ‘a surface indication that usually corresponds to a deeper problem in the system’. Given the number of code smell detectors already available, it can be difficult to reason about which one may be the best for one’s specific use case. For this reason, we have decided to implement a small, but easily extendible, command-line python tool called PyReflect that will enable visual linting of Java code for a few code smells such as duplicated code, god class, long method, long parameter list, and lazy class. PyReflect also has certain added enhancements such as user defined rules, simple refactoring suggestions to remove these smells, and a visualization model to represent the Java code in a simplified manner. We also evaluated PyReflect against other code smell detection tools such as JDeodorant and PMD to conduct a comparative study. Our results show that in terms of configurability, PyReflect may be easier to use and can detect the most common code smells whereas PMD is more complex and tries to detect over 300 code smells which can be too specific to the novice user. While JDeodorant may be similar to PyReflect in terms of ease of use, it is limited in the number of code smells it can detect. Overall, we provide a detailed discussion on which tools would be best to use depending on the situation.

1. INTRODUCTION

Martin Fowler, one of the founding leaders in code refactoring, refers to a code smell as ‘a surface indication that usually corresponds to a deeper problem in the system.’ The presence of code smells usually indicates a need for refactoring. He goes on to warn that disregarding code smells can cause programs to become unmanageable and harder to debug over time. Automated or semi-automated code smell detection tools can help programmers determine bad coding practices when code reviews can no longer be done manually due to the large size of the program’s code base. Given the fact that there are already a number of complicated linting and static code analysis tools out there, it can be difficult to reason about which one may be the best for one’s specific use case. For this reason, we have decided to replicate a more lightweight and flexible code smell tool and evaluate it against existing detectors on functionality via a comparative study. A better understanding of existing code smell detectors can help developers determine what future tools are most effective in understanding programmers’ needs and accelerating their code development workflow.

Our goal for this project is to implement a small, but easily extendible, command-line python tool called PyReflect that will also enable visual linting of Java code for a few of the targeted code smells. We chose to implement our tool using Python due to the fact that Python is a simple and lightweight programming language which often makes programs more efficient to run. However, we chose to evaluate our tool on Java code because many of the existing code smell detectors are already targeted towards Java code. Our targeted smells are duplicated code, god class, long method, long parameter list, and lazy class. In short, duplicated code means that the same code structure appears in more than one place. God class is a program that performs too much work on its own and contains too many instance variables or methods. Long method is a method that is too long which makes it difficult to understand or extend. Long parameter list occurs when a parameter list is too long and is difficult to understand. Lazy class occurs when a class does not a lot of work and may have too few methods. We explore these paradigms in more detail in later sections. Furthermore, we added certain enhancements to our tool such as user defined rules, simple refactoring suggestions to remove these smells, and a visualization model to represent the Java code.

1.1 Structure

The paper is organized as follows. Section 2 goes over related works that have been done on already existing code smell detection tools. Section 3 goes over our approach in how we implemented PyReflect. A large component of the project will be comparing existing code smell detection tools - particularly JDeodorant and PMD, with our tool. We chose to evaluate PyReflect with these particular code smell detection tools as both detect code smells in Java programs. Furthermore, JDeodorant and PMD both look to detect code smells as the ones that we mentioned in the Introduction. Section 4 will compare and contrast these linting

WOODSTOCK ’97 El Paso, Texas USA
© 2016 ACM. ISBN 123-4567-24-567/08/06...$15.00
DOI: 10.475/123_4

tool results with our tool. Section 5 goes over our visualizations and PyReflect website, while Section 6 covers some of the validity threats that could be presented toward this study. Lastly, we will discuss our conclusions from the comparison and explore potential opportunities for future work.

2. RELATED WORK

Analyzing the performance of various commercial and academic tools is not necessarily a new or innovative task. However, we are doing this in the context between both existing programs and the new program we propose: PyReflect. We compare these code smell detectors on their ability to detect ground truths in files of varying complexities. In the following paragraphs, we discuss some existing papers in order to set a baseline for some of the common evaluation criteria.

"Automatic detection of bad smells in code: An experimental assessment" [1]. When code smells are found in programming code, it is often an indication that refactoring needs to done otherwise the code can be harder to maintain overtime. Existing code smell detection tools are used by programmers when code size because unmanageable manually. Furthermore, since code smells are defined subjectively, it can be hard to evaluate and compare different code smell tools. This paper compares four different code smell detectors such as JDeodorant, inFusion, PMD, and Checkstyle. These tools are selected for evaluation mainly due to the fact that they are downloadable, actively maintained, and results are easily accessible. These tools are evaluated against different versions of an open source Java project, GanttProject. It determines the strengths of these tools and what improvements can be made to bring efficient code smell detection tools in the future. This paper produces an assessment of the agreement, consistency, and relevance the answers produced. Their experiment determines that different code smell detectors do not agree for all code smells detected but they do detect problematic code that needs to be fixed in the future.

"JDeodorant: identification and application of extract class refactorings" [2]. Evolutionary software changes in object oriented programs often result in large and complex classes which can be identifiable as 'god classes'. This paper proposes a way to detect god classes and provides refactoring options which are implemented in JDeodorant. Instead of directly defining the god class metrics, it defines god class as a class that can be decomposed into other classes and employs a clustering technique that can identify opportunities of extract-class refactoring.

"Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems" [3]. This book looks to demystify the design metrics used to assess the size, quality, and complexity of object-oriented software systems based on statistical information gathered from live industrial projects. Figuring out the accepted semantics for design can be difficult to do in practice. However, once the book proposes definitions for these metrics, it provides strategies to detect these common code smells (such as god class, feature envy and data class). In recent years, some of these strategies have been adopted by well-known code smell detectors like PMD, iPlasma and inFusion.

"Magnify - a new tool for software visualization" [4]. This paper introduces Magnify, which is a Java static code visualization tool. This paper discuss the design of the tool and present visualizations of sample open-source Java projects of various sizes. As modern software programs get more and more complex, maintenance can be difficult if documentation is not up to date. Furthermore, dependencies can be hard to discover by only looking at the source code. By creating a software visualization tool, it can be easier to understand code. Magnify first parses the source code, identifies dependencies, and records this information in a graph based data model. The nodes represent the methods, classes, and packages while edges represent dependencies and the hierarchical structure. Various colors are also used to represent the quality, sizes, relationships between nodes. Magnify is also always up to date since it automatically generates from the current software.

3. APPROACH

To avoid rewriting a Java parser, since many of these parsers already exist, we use the python library PlyJ to parse and generate Abstract Syntax Trees (AST) for the underlying java files. The PlyJ module returns a traversable Abstract Syntax Tree object that represents the given Java source file.

In addition to code smell detection, the PyReflect module gives users the ability to translate their multi-file Java package into a class/file dependency tree. Since we wanted to create a visualization of this Java program using D3.js, we traversed through the tree object and represented it as a JSON object. A rendered dependency/class visualization of a program will enable developers and architects to verify and compare their intended program designs with what is actually parsed in the implemented source code. These visualizations are interactive and permit incremental visual expansion of the individual nodes. For example, the graphs can be lightly manipulated and modified.

As enhancements to our tool, users can define their own code smell parameters into PyReflect. As an example, a user can determine a long method by specifying the maximum number of statements a method must contain in order for it to not be considered as a code smell. Furthermore, simple refactoring suggestions such as rename, move, extract are provided to the user for each code smell detected. PyReflect is run directly from the command line, with the different smell detectors used by specifying different command options (or -a to run all the code smells simultaneously on your source project).

3.1 Detected Code Smells: Definition and Detection

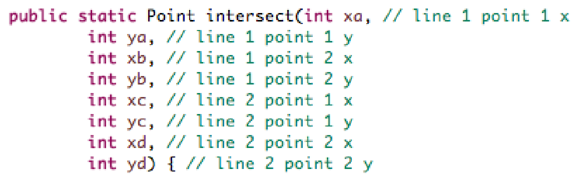
The command line version of PyReflect investigates the following: long method, long parameter list, god class, lazy class, and duplicated code. These code smells will be the basis for which we compare the existing tools. In order to perform the comparison reliably, we define these terms precisely below.

3.1.1 Long Method

A long method is a method that has an unusually large number of program statements and is possibly difficult to understand, change, or extend by others. Long methods can be detected by counting the number of lines of code (NLOC) and checking if it is larger than a given threshold (in the case of PyReflect this parameter can be easily specified). Long method detectors will usually ignore whitespace and blank lines, counting only full code statements - providing a more accurate representation of the method length. Both PMD and our tool, PyReflect's long method detection strategy uses the above rules. JDeodorant, however, uses a slicing technique to determine if a class is eligible for extract method refactoring [5].

3.1.2 Long Parameter List

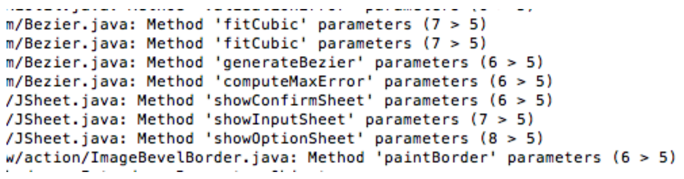
A long parameter list occurs when a parameter list is too long and thus difficult to understand. Methods with numerous parameters are a challenge to maintain, especially if most of them share the same data type. These situations usually denote the need for new objects to wrap the numerous parameters. It can be detected by counting the number of parameters of each method and checking if it is larger than the given threshold or not. Both PMD and our tool, PyReflect use such way to detect long parameter list while JDeodorant does not support detection of this smell. Long parameter lists could be difficult for maintenance in the future by reducing modularity. It also reduces code reusability because a particular function might be too long and specific to be used in other contexts. With the pyreflect -lp N option, the user is able to quickly identify functions across their entire Java project that have more than N parameters. The program will show you the exact function and class file where the violations occur. This can be done with a single command.



```
public static Point intersect(int xa, // line 1 point 1 x
        int ya, // line 1 point 1 y
        int xb, // line 1 point 2 x
        int yb, // line 1 point 2 y
        int xc, // line 2 point 1 x
        int yc, // line 2 point 1 y
        int xd, // line 2 point 2 x
        int yd) { // line 2 point 2 y
```

Figure 1: Example of a method with reasonably high parameter count

While it may seem that all of the function parameters are being used in Figure 1 above, abstracting these parameters into Point class objects may be better aligned with Object-Oriented principles. Long parameter list checks can help programmers find opportunities for these kind of refactorings. The output of PyReflect and other tools will mark all such cases of functions with long parameter lists according to the user's preferences.



```
m/Bezier.java: Method 'fitCubic' parameters (7 > 5)
m/Bezier.java: Method 'fitCubic' parameters (7 > 5)
m/Bezier.java: Method 'generateBezier' parameters (6 > 5)
m/Bezier.java: Method 'computeMaxError' parameters (6 > 5)
/JSheet.java: Method 'showConfirmSheet' parameters (6 > 5)
/JSheet.java: Method 'showInputSheet' parameters (7 > 5)
/JSheet.java: Method 'showOptionSheet' parameters (8 > 5)
w/action/ImageBevelBorder.java: Method 'paintBorder' parameters (6 > 5)
```

Figure 2: Figure showing violations of a long parameter test with threshold value 5.

This a preview of the long parameter results on the JHotDraw codebase - the long parameter test will extract and print all the methods that violate the specified constraint.

3.1.3 God Class

A god class is typically considered a centralized location of the code system intelligence. At first glance, it may seem like this is a good thing - a majority of the functionality of the whole program is in one central location; however, in practice this usually presents a few problems. A god class results in tight coupling between all the components in the program and reduces modularity. Often if you need to extend or modify the program later, it may not be easily doable because of the overly convoluted design of a present god class. These classes are usually indicated by performing too much work on their own and using too much data from other classes. Lanza and Marinescu propose a way to detect god class in a codebase using these quantifiers:

Weighted Method Count (WMC): the sum of statistical complexity of all methods is very high.
Access to Foreign Data (ATFD): the number of external classes.
Tight Class Cohesion (TCC): the relative number of methods directly connected via accesses of attributes.

According to Lanza's book, *Object-Oriented Metrics in Practice* [3], a class is identified as god class if the following conditions hold on these quantifiers:

$$WMC > VeryHigh \tag{1}$$

$$ATFD > Few \tag{2}$$

$$TCC < \frac{1}{3} \tag{3}$$

In practice, it's possible that the Few and VeryHigh thresholds are not strictly defined, or at least not uniform in all projects. Both PMD and our tool, PyReflect, adopt the above rules. In contrast, JDeodorant identifies a class as a 'god class' if it can likely be decomposed into other classes and has a high opportunity of extract class refactoring [6].

3.1.4 Lazy Class

A lazy class is a class that attempts to do too little. This is usually indicated by the number of methods it has. Lazy classes often offer opportunity for refactoring or merge, in that they could potentially be absorbed by other classes without introducing complexity. In this sense, an additional class file can be removed from the project, without affecting the overall complexity of the project and perhaps reducing it if the correct refactoring merge is done. It can be difficult to quantify the false positive or true positive rates on this code smell as it is heavily dependent upon the implementation (i.e. the class' minimal nature is intended by the programmer's design). As stated earlier, the lazy class detector can be useful for detecting smaller classes that may be unnecessary on their own.

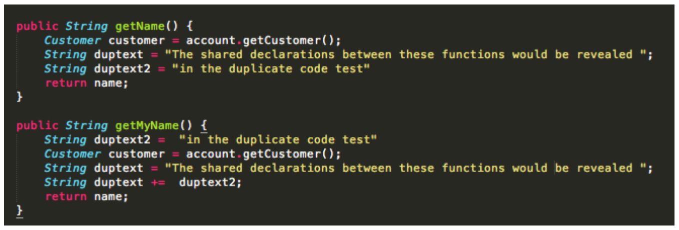
3.1.5 Duplicate Code

We consider a method's code to be duplicated if the same or similar code structure appears in more than one place. This can be exact duplication or duplication tolerating order rearrangements. Many tools only allow direct ordered line mappings for determining code similarity which can be quite limiting. Our tool will enable less tightly restricted comparisons of methods by looking for methods that contain similar commands that may be refactorable even if the individual statements in the method appear out of order.

There are many allowable definitions for what defines a code clone. According to materials provided in the code smells course by Jason Gorman [7], duplicate code is defined in the context of the same method.

For example, PMD reports duplicate code smell when it finds a portion of code that is duplicated at least once and that is composed of at least 100 tokens. JDeodorant itself does not support duplicate code smell detection but provides visualization and refactoring suggestions based on the detection results from other tools like CCFinder and CloneDR [8].

Our tool, PyReflect, compares the similarity between different functions. Given a threshold value, PyReflect return a duplicate code smell if the number of similar statements exceed that particular threshold value. Furthermore, our tool does not require these similar statements to be in the same order. Statements that match but are in different order will also be considered as duplicated code. We show an example of this in the figure below.



```
public String getName() {
    Customer customer = account.getCustomer();
    String duptext = "The shared declarations between these functions would be revealed ";
    String duptext2 = "in the duplicate code test"
    return name;
}

public String getMyName() {
    String duptext2 =  "in the duplicate code test"
    Customer customer = account.getCustomer();
    String duptext = "The shared declarations between these functions would be revealed ";
    String duptext +=  duptext2;
    return name;
}
```

Figure 3: Figure showing violations of a long parameter query with parameter length threshold value of 5.

While this is somewhat a contrived example, it is possible that repeated behavior is performed in multiple functions within a class. This can be especially likely in larger programs with multiple authors. One limitation of PyReflect is that it does not try to reason about the semantics of the language or how the variables are used, only the code structure. For example, if we have a class where two variables and functions are used in the same way, but the names are different - the duplicate code test would not report a positive result in this case. PyReflect performs a cross-checked textual comparison that ignores statement order. Another example of this is shown later in the evaluation section.

4. EVALUATION

We evaluate PyReflect in the context of a variety of different code datasets, ranging from simple single-class projects to very complex. The simple datasets serve to as a set a benchmarks for PyReflect, while we use the real world Java project (JHotDraw) as the benchmark for the performance comparison between PyReflect and other Java static analysis tools.

4.1 Datasets

Firstly, to evaluate the effectiveness of our tool in detecting the code smells mentioned above, we use simple labeled datasets designed for code smell detection from Jason Gorman's online workshop [5]. This workshop introduces novices to the fundamental practices of refactoring and focuses on building good instincts for spotting common code smells. This dataset contains 15 small examples of code smells where each example has on average 50 lines of code. We chose these examples because all examples provided are simple, well organized and labelled which can also be used for debugging during the development process.

Secondly, we also use a medium scale test case [9] that has a combination of multiple code smells and will use it to see how our tool works in detecting multiple smells in one scan. This dataset is a single folder containing 8 java files that range from 15 to 90 lines of code.

Finally, we compare our tool, PyReflect on the grounds of the smells in a real-world program–JHotDraw. JHotDraw is a drawing editor program written in Java and is used as the benchmark test case by JDeodorant for god class detection. This is the reason why we chose it to evaluate the effectiveness of our tool and also compare the performance of tools mentioned above in terms of completeness, speed of execution, configurability, and limitations with respect to the target code smells.

Table 1: JHotDraw Project Statistics

Number of lines	32,126
Number of methods	3,377
Number of files	309
Size of entire project	23.9 MB

For each code smell mentioned above, we will answer the following questions in the context of PyReflect, PMD, and JDeodorant:

1. What is the number of bad smells found in JHotDraw by those tools?
2. Do the test results agree with each other? If not, why do they disagree?
3. Are there any significant differences in performance?
4. What were limitations of the tools?
5. Are the results useful?

In order to determine the number of bad smells found in JHotDraw, we ran PyReflect, JDeodorant, and PMD simultaneously to see if these tools can detect certain code smells such as long parameter list, long method, god class, lazy class, and duplicate code all at the same thresholds. To check if the results agree with each other, we recorded the number of code smells each of these methods found and manually checked to see how many of these code smells agree with each other. In order to determine performance of each of these three methods, we checked the time it took to run each tool on JHotDraw. After we got our results, we also discussed the limitations and benefits of each tool. We based the usefulness of our results on whether we can make conclusions regarding the advantages and disadvantages of each code smell detection tool.

4.2 Existing Static Code Smell Detectors

PyReflect is certainly not the first static code analysis tool written for Java. Since many code smell tools already exist, we have chosen to evaluate two of the more well-known static analysis tools that look at code structure: PMD and JDeodorant.

4.2.1 *JDeodorant*

JDeodorant [1] is an Eclipse plugin that is designed to detect common code smells, such as god class, feature envy, long method and type checking. Unlike other code smell detection tools, JDeodorant focuses on providing refactoring recommendations and it uses the possibility of applying a specific kind of refactoring to detect the corresponding code smells.

4.2.2 *PMD*

PMD is another source code analyzer and also provides an Eclipse plugin version. PMD is able to detect large class, long method, long parameter list, and duplicated code. Furthermore, it can also detect over 300 potential problems, such as dead code, empty try, unused variables. It also allows users to select rules from a given list and to set the threshold values of the evaluation metrics.

4.3 Performance Comparison

For the larger performance comparison, we ran each of the listed code smell tests for PyReflect, PMD, and JDeodorant on the JHotDraw Project. We note that each has different default thresholds to trigger a detection flag, which result in the differences. Table 2 shows our findings for running the various tests on the project, and the differences between the results are explored in the following sections. Note that a value of 'N/A' in the table indicates that the tool does not support that particular code smell test.

4.3.1 *Long Parameter List*

Long Parameter lists can be detected simply by counting the number of parameters of each method. JDeodorant does not support this smell while PMD and PyReflect use exactly the same approach. When we set the threshold to be 7, the results of PMD and PyReflect agree with each other as expected. In other long parameter list cases, it is better to wrap the information in one object which makes it easier to understand and maintain.

4.3.2 *Long Method*

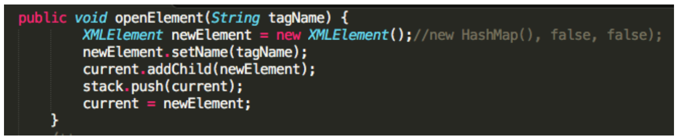
Each of these three tools use different approaches for determining long method code smells. In its search for long method, JDeodorant includes all the methods that it finds an extract method refactoring opportunity, independently of their length [8]. The result shows that in JHotDraw, JDeodorant finds no method that need extract method refactoring. However, both PMD and PyReflect detect long method based on their length. The only difference is that PyReflect ignores whitespace and brackets when counting the number of lines. Therefore, when setting the threshold to 75, we noticed that PMD's positive answers include all the positive answers of PyReflect. In terms of long method, the approach that our tool uses is more strict but more reasonable than that of PMD.

4.3.3 *God Class*

In comparison to PyReflect and PMD, we observe that JDeodorant uses a significantly different strategy for detecting god classes; JDeodorant performs clustering on methods and attributes whereas PMD and PyReflect, however, calculate metrics for access to foreign data and class functional complexity. 20 out of the 57 positive answers of JDeodorant are included in that of PMD. However, the results of PMD and PyReflect are similar since they theoretically use the same method but have different ways of implementation.19 out of the 26 positive answers of PyReflect agrees with the results of PMD.

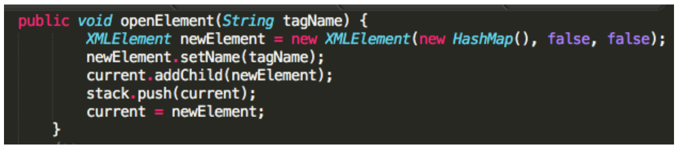
4.3.4 *Duplicate Code*

As mentioned above, JDeodorant does not support duplicate code but instead provides refactoring recommendations based on the result of other tools. When setting the threshold to 800 tokens, PMD reports 3 duplications. Then, after further inspection into these locations, we found obvious evidence for copy-paste as shown in the codes below. Compared to PMD, the technique we use is much simpler but less powerful. When searching duplicates, PyReflect does not scan the programs across each of the files, instead it is only able to consider programs in the same file. We also noticed that one of the two positive answers it reports is actually an overloading case rather than duplication which is the other limitation of PyReflect. Therefore, in terms of duplicate code detection, PMD is more powerful than our tool and the results it generates do imply the bad copy-paste smell according to our investigation. But one weakness of PMD is that the duplicate detection is not integrated in its ruleset and we need to launch a separate module. For context, we present an example occurrence of duplicate code that was flagged from PyReflect's analysis of duplicate code below.



```
public void openElement(String tagName) {
        XMLElement newElement = new XMLElement();//new HashMap(), false, false);
        newElement.setName(tagName);
        current.addChild(newElement);
        stack.push(current);
        current = newElement;
    }
```

Figure 4: First occurance of method openElement function presented in the project



```
public void openElement(String tagName) {
        XMLElement newElement = new XMLElement(new HashMap(), false, false);
        newElement.setName(tagName);
        current.addChild(newElement);
        stack.push(current);
        current = newElement;
    }
```

Figure 5: Duplicated Method content that occurred later within the same project file

4.3.5 *Lazy Class*

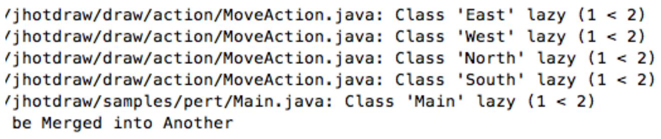
Lazy class is a functionality that is not offered by either JDeodorant nor PMD, though is offered in PyReflect. Lazy class is recognized as a legitimate code smell. Running PyReflect on the JHotDraw codebase reveals several classes that are minimal in nature. Here's a short snippet of output of running PyReflect on the JHotDraw module.

It may or may not be better to refactor these classes by merging them with others - such merges can often reduce the complexity of larger projects. For example, the Main class listed only has one "main" method in it, though this main method could be refactored into one of the other pre-existing classes. West, East, North, and South are all single-method subclasses. In the case of a class that's lazy that is also a subclass, it is often possible to use a collapse hierarchy method, or turn that class into an inline class [10].

Table 2: Number of Detected Code Smells on JHotDraw Project

	Long Parameter List	Long Method	God Class	Duplicate Code	Lazy Class
JDeodorant	N/A	0	57	N/A	N/A
PMD	4	30	42	3	N/A
PyReflect	4	3	26	2	11

'N/A' indicates test not supported



```
/jhotdraw/draw/action/MoveAction.java: Class 'East' lazy (1 < 2)
/jhotdraw/draw/action/MoveAction.java: Class 'West' lazy (1 < 2)
/jhotdraw/draw/action/MoveAction.java: Class 'North' lazy (1 < 2)
/jhotdraw/draw/action/MoveAction.java: Class 'South' lazy (1 < 2)
/jhotdraw/samples/pert/Main.java: Class 'Main' lazy (1 < 2)
 be Merged into Another
```

Figure 6: Snapshot of Lazy Classes discovered during project analysis of JHotDraw

4.4 Discussion

God class is certainly the most complex of the listed code smells, and unless obscene, god classes can also be hard to identify by the programmers' themselves. Often god classes are unknowingly created because of the human tendency to centralize all the functionality for ease in locating certain methods. Unfortunately, this doesn't scale at large. In this discussion, we performed a more precise time analysis on the complexity of god class extraction on the same program.

Based on the results and analysis above and the time consumed for detecting God Class, we can discuss the strength and weakness of each tool's parsing ability and result generation.

Table 3: Time Consumed for Detecting God Classes

	JDeodorant	PMD	PyReflect
Parsing	25.46s	17.34s	14.85s
Detecting	85.67s	1.56s	1.60s

We discover that the detection strategy that JDeodorant uses is completely different from PMD and PyReflect. Instead of calculating characteristics directly from the project, it investigates the refactoring opportunities and reports bad smells if such an opportunity is high. In addition, JDeodorant also provides refactoring suggestions which can be applied automatically. Overall, the emphasis of JDeodorant is more towards how we can refactor code instead of purely detecting bad code smells. The weakness of JDeodorant is that the number of smells it supports is quite limited and that it can only run one smell detection each time. And due to the high complexity of its detection techniques, it consumes much more time compared to PMD and PyReflect.

The detection strategy PMD uses is quite similar to ours and its performance is even better in terms of god class and duplicate code. But the weakness of PMD is also obvious. Firstly, PMD supports over 300 code smells and some of them like "variable name is too long/short" seem not useful actually. What's more, it can be overwhelming to the user to determine where to start looking. Secondly, the way PMD names each code smell is different from JDeodorant and PyReflect. For example, it names long method as "excessive method length" which makes it even harder for us to locate the one we are looking for. Finally, every time we change the ruleset or the parameters, PMD needs to be rebuilt.

Our tool, PyReflect employs similar techniques to PMD and performs better in terms of detecting long method. Unlike PMD, PyReflect is run directly from the command line, with the different smell detectors used by specifying different command options and there is no rebuilding needed. Also, the smells it supports are those most commonly used. Overall, our tool is much easier to use compared to PMD. Compared to JDeodorant, PyReflect supports a larger but reasonable number of code smells and runs much faster. However, there are still some limitations especially for duplicate code detection which makes PyReflect less powerful compared to PMD.

Generally speaking, it is not trivial to determine which tool is the best. If you want automatic refactoring applied for a specific smell, JDeodorant is a better choice. However, if you just want to focus on the detection of some common code smells, our tool should be better and easier to install and use. If you care about detecting more than these common smells and want to detect other bad smells, there is a high possibility that PMD has already included it in its ruleset.

5. VISUALIZATIONS

The approach to the visualization component of PyReflect was around ease-of-use. There are many existing Java source code visualization tools available in open source, however we found these to be in some cases too complicated to try to parse and understand. From a development standpoint, a good source code visualization should tell you some high level characteristics about your program that would not be easy to extract or find quickly otherwise.

PyReflect can be run with a -p option, which will generate a unique source JSON file tree for each Java program that it discovers recursively within the command's original execution directory. The resulting trees generated extract variable, method, and other class declarations in each file and render them in a traversable tree structure. These tree structure tabs can be saved, printed and shared, or developed and used as part of a high level class description to guide developers in terms of what the classes it should contain.

These JSON files can be rendered through the website, or via text upload at the following url: http://chrisbuonocore.com/cs230project/

Magnify is another software visualization tool that works in a similar vein to PyReflect. Magnify parses Java source files and searches for dependencies between classes and objects in a nodal form. The following diagram shows a visualization of the Java Play framework where the biggest

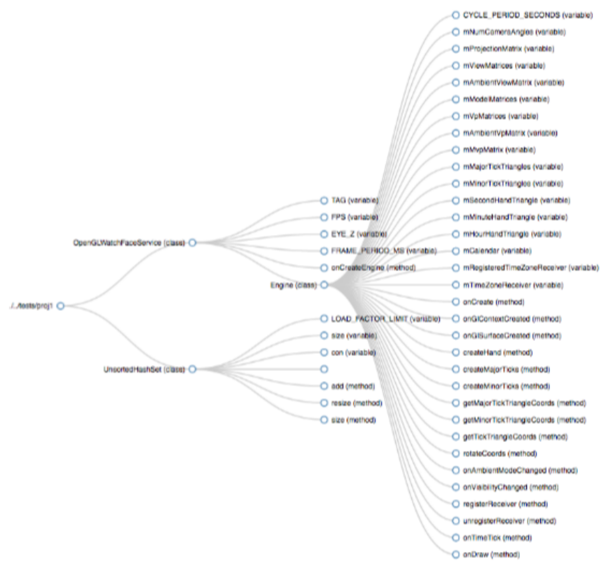

./../tests/proj1
OpenGLWatchFaceService (class)
UnsortedHashSet (class)
TAG (variable)
FPS (variable)
EYE_Z (variable)
FRAME_PERIOD_MS (variable)
onCreateEngine (method)
Engine (class)
LOAD_FACTOR_LIMIT (variable)
size (variable)
con (variable)
add (method)
resize (method)
size (method)
CYCLE_PERIOD_SECONDS (variable)
mNumCameraAngles (variable)
mProjectionMatrix (variable)
mViewMatrices (variable)
mAmbientViewMatrix (variable)
mModelMatrices (variable)
mVpMatrices (variable)
mAmbientVpMatrix (variable)
mMvpMatrix (variable)
mMajorTickTriangles (variable)
mMinorTickTriangles (variable)
mSecondHandTriangle (variable)
mMinuteHandTriangle (variable)
mHourHandTriangle (variable)
mCalendar (variable)
mRegisteredTimeZoneReceiver (variable)
mTimeZoneReceiver (variable)
onCreate (method)
onGlContextCreated (method)
onGlSurfaceCreated (method)
createHand (method)
createMajorTicks (method)
createMinorTicks (method)
getMajorTickTriangleCoords (method)
getMinorTickTriangleCoords (method)
getTickTriangleCoords (method)
rotateCoords (method)
onAmbientModeChanged (method)
onVisibilityChanged (method)
registerReceiver (method)
unregisterReceiver (method)
onTimeTick (method)
onDraw (method)

Figure 7: Example PyReflect Tree Visualization of Two Class Objects

node represents the project root. The brightly red packages reveal potentially high complexity of other classes.

PyReflect, in contrast, operates as more of a static code visualizer that illustrates all the object declarations and usages within relevant functions and classes. It does not try to make assumptions or reason about complexity. These assumptions are often subjective and are left to be determined at the programmer's individual discretion.

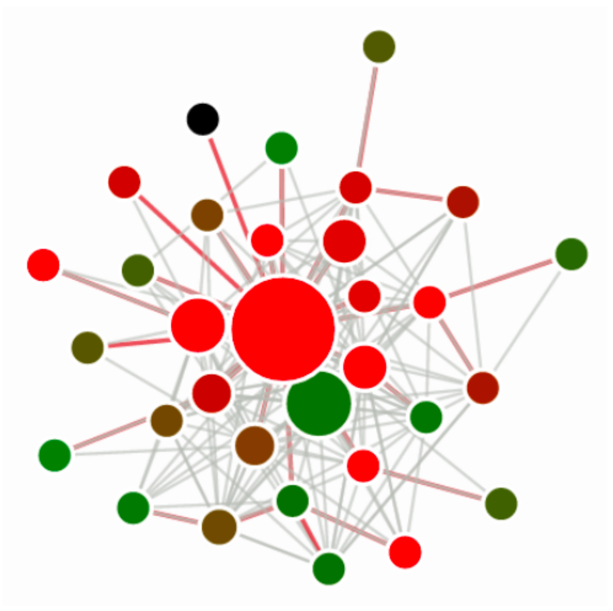


Fig. 5. The visualization of Play 1.2.5 produced by *Magnify*

Figure 8: Magnify's Unlabeled Class Dependency Graph on Java *Play* Framework

6. THREATS TO VALIDITY

There were a few threats to validity that we had to consider while both implementing the PyReflect tool, and comparing it to the outside modules JDeodorant and PMD. One of the first points we considered is whether PyReflect's parsing model can easily extend to other languages or is it just relevant to just Java. While we used Java as our target language to analyze, the code smell model we use is not restricted to Java; PyReflect uses the Java parsing tool PlyJ to generate a source code AST for the program files. Given a parser and AST of similar form for another language, the methods PyReflect use could be adapted for the new language's AST models.

We considered the fact that if our examples in our evaluation dataset is legitimate enough to be used as benchmarks for the indicated code smells. For example, we wondered if there is a universal, or clear enough, definition of all the code smells in order to be automatically detected or if code smell detection intrinsically require human judgement or review. Each individual code smell may not have universally agreed upon definition, but it could be argued that code smells tend to contribute to code decay and difficulty of maintenance should they be left unaddressed. Where rigid or obvious, such as the long parameter list or long method, definitions are fairly standard. If not, we followed the definitions provided in the previous code smells section (based on a blend of the sources offered in the references section) for modeling our PyReflect code smell detection program.

Our comparison uses JDeodorant and PMD Java static analyzers as a benchmark for PyReflect's performance. One question could be whether JDeodorant and PMD are acceptable to be treated as standard code smell detectors used for Java programs. There are a number of tools available out there, though PMD and JDeodorant fall in the same category of static analyzers as PyReflect, and also contain most of the same code smells we evaluate against. The comparison is against tools that use the same code smell analysis strategy. Because of the categorical nature of these tools, we thus consider this to be a weak threat to our original testing.

Another question we ask is whether PyReflect's visualization model scale for programs of very large size. PyReflect's visualization mechanism is currently at a fairly young stage - the tree structure could definitely be adapted to work better for larger programs. For now, PyReflect will look to render each program's source code as a Class "Declaration" tree. This tree reveals all the instantiated objects and variable dependencies that are introduced within each class. The problem is scalability, however, these generated trees may grow to be very large and may lose their ease of understanding by the end user should the target source code scale to thousands or millions of lines. A different visualization could potentially be explored in the future that is able to present well - even under large codebases.

7. CONCLUSIONS

Constructing new code smell tools, let alone performing full code smell tool analysis, is a task that can take many years to perfect. For example the open-source python code smell detection tool, Pylint, has been developed and maintained over a decade since its first introduction [11]. Pylint focuses on enforcing style conventions (such as spacing, naming, usage, etc.) in the context of python programs.

We implemented the new code smell detection tool, PyReflect, with a simpler interface and support for a few well-known code smells such as duplicated code and god class. We also added certain enhancements to our implementation such as user defined smell threshold values, simple refactoring suggestions to remove these smells, and a class declaration visualization to represent Java source code. In addition, we allow support for complex multi-file Java programs by simply pointing the PyReflect program at a single project folder; PyReflect will automatically parse and analyze all source code files in that folder for the user. After implementing PyReflect, we evaluate it against a few other well-known code smell detection tools such as JDeodorant and PMD via a comparative study.

Our results show that PyReflect is effective if users want a simple tool to discover the most common types of code smells. In comparison, PMD has a more complicated interface and can discover over 300 types of code smells. While impressive, it's possible that this can be overwhelming to a more novice user and may even stop adoption. We also noticed from the study that without changing the defaults manually and precisely on PMD and JDeodorant, many of the simple code smell datasets could not be successfully marked.

Although JDeodorant is simpler, it can be quite limiting in the number of supported code smells. JDeodorant has the benefit of being able to provide concrete action plans for refactoring many types of common code smells. Each code smell detection tool has its advantages and disadvantages. Knowing when to use each tool can be difficult to determine in general, but given a specific situation, one may be more effective than the others.

In this paper, we have shown the limitations and strengths of each to better inform those who stand to benefit from such tools.

8. FUTURE WORK

There are definitely many potential extensions of this project that could continue beyond this initial effort. We present these potential enhancements and leave the implementation and details to the next developer's tastes. In terms of future work, PyReflect could be expanded to support additional code smells, and perhaps provide more scalable visualizations for larger programs. One thing that often inhibits adoption is often complexity of configurability of such a detection tool; however, it would be beneficial to allow the user to define their own smell detection templates and evaluate their code based on these templates (to fit the individual/specific needs of the user). Furthermore, we would like to perform a more-detailed performance comparison of the different code smell detection tools. In this initial evaluation, we compare the five different code smells on one real world project, JHotDraw. We selected to analyze the smaller and medium code smell data sets to ensure core functionality of the tool, with the final real-world dataset to be the program JHotDraw. While found several significant performance contrasts between the results of the different tools on this program, it would be beneficial in the future to compare across additional real-world programs for additional confidence. Finally, we would like to add exportable and editable visualizations of the program tree structure, for example, allowing edits to the underlying code and API's directly through the visual diagrams.

9. REFERENCES

[1] Fontana, F. A., Braione, P., & Zanoni, M. (2012). Automatic detection of bad smells in code: An experimental assessment. Journal of Object Technology,11(2), 5-1.

[2] Fokaefs, M., Tsantalis, N., Stroulia, E., & Chatzigeorgiou, A. (2011, May). JDeodorant: identification and application of extract class refactorings. InProceedings of the 33rd International Conference on Software Engineering(pp. 1037-1039). ACM.

[3] Lanza, M., & Marinescu, R. (2007). Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems. Springer Science & Business Media.

[4] Cezary Bartoszuk et al. (2013). Magnify - a new tool for software visualization.

[5] Tsantalis, N., & Chatzigeorgiou, A. (2011). Identification of extract method refactoring opportunities for the decomposition of methods. Journal of Systems and Software, 84(10), 1757-1782.

[6] Fokaefs, M., Tsantalis, N., Chatzigeorgiou, A., & Sander, J. (2009, September). Decomposing object-oriented class modules using an agglomerative clustering technique. In Software Maintenance, 2009. ICSM 2009. IEEE International Conference on (pp. 93-101). IEEE.

[7] Jason Gorman Refactoring Workshop.
http://www.codemanship.co.uk/refactoring.html

[8] Mazinanian, D., Tsantalis, N., Stein, R., & Valenta, Z. (2016, May). JDeodorant: clone refactoring. In Proceedings of the 38th International Conference on Software Engineering Companion (pp. 613-616). ACM.

[9] Persa, Dan, Code Smell, (2014), GitHub repository,
https://github.com/danpersa/code-smell

[10] Lazy Class Smell. - CSSEMediaWiki.
http://wiki3.cosc.canterbury.ac.nz/index.php/Lazy_class_smell

[11] Pylint Code smell detection and linting tool.
https://github.com/PyCQA/pylint

APPENDIX

.1 How to Download and Run PyReflect

The main source of documentation for the PyReflect module is via the Github, located at https://github.com/cbonoz/cs230project. The README.md file there will contain the complete installation instructions for the command line tool and website. You will need at least Python 2.7 and the PlyJ python library (installable via a python package manager like pip). The installation process will add a PyReflect alias command to your terminal without modifying your path, so you can invoke code smell detection on any Java project you have via without restriction of directory location. To get started after installing, try entering 'pyreflect -h' from the command line, or navigating to the downloaded project folder and running 'python pyreflect -h' directly from the project folder.

.2 How to Use the PyReflect Website

In order to use the PyReflect Visualization tool locally, you need to download node/npm and bower. Once this is done, gather generated project JSON files and load your tree files into the project's app/trees folder. Finally, start a localhost server to view the visualization web page (gulp serve from root folder would work for this). The Visualization tool can also be accessed directly on the web via: http://chrisbuonocore.com/cs230project/. Here you can try rendering some example PyReflect Java tree files on the server, and also render your own PyReflect source trees as JSON objects.