The Smell of Poor Design

Remco de Man
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
r.j.p.deman@student.utwente.nl

ABSTRACT

In programming education, tools which can help provide quality feedback with respect to program design are needed. Most novice programmers write code that contains design smells which originate from the programmer not understanding important design concepts. Tools that can detect problems in application design could serve two purposes. First of all, it helps to define design requirements for applications. Secondly, since giving individual feedback to every novice programmer is very time-consuming, this could lower the pressure on programming educators. This study will explore design guidelines for Processing code. By analyzing sketches, design smells will be documented and tools will be proposed for detecting design smells in these sketches.

Keywords
Software smells, code smells, design smells, programming education, software design

1. INTRODUCTION

Programming has become an increasingly important subject for many different disciplines. First, programming was only important for technical disciplines. Nowadays, also novices in less technical disciplines learn some programming. Giving quality feedback to these programmers is especially important since, for these novices, programming is often difficult. Since giving individual feedback increases pressure on programming educators, research to novice programmer code and tools to give automated feedback to this code has become more important in the last few years.

An important part of programming education is application design. Choosing the right design while programming an application is important for clear understanding of the code. An experiment with the block-based Scratch language has shown that badly written code can lead to decreased system understanding by programmers themselves but also for more experienced programmers that review the code[4].

While many novice programmers are able to write code that effectively solves a problem, they have trouble understanding the principles of a good application design. Although this does result in working code, it does not fit the design principles of object-oriented standards. This code contains so-called design smells, which are symptoms of not understanding the deeper design paradigms behind the programming language.

In order to make novice programmers aware of design smells and good application design, guidelines for application design should be provided. These guidelines can be used to give quality feedback on the code of novice programmers as well as helping them understand the rules of application design.

In the recent years, lots of research has been performed on automated feedback frameworks using static code analysis, which primarily look at styling issues and possible bugs. Although successful in providing feedback on these aspects of the code, they give almost no feedback on the application design. Feedback generated on application design in existing feedback tools is mostly based on software metrics, which are understood by novice programmers. As such, novice programmers will not understand the feedback and will not gain a deeper understanding of object-oriented application design.

If the concepts of application design are well defined, it is possible to make feedback tools aware of these concepts. This allows feedback tools to detect design smells in novice programmers code and give useful feedback on them. This will help novice programmers to write better code while also gaining a deeper understanding of the application design and object-oriented programming patterns in general.

This study will focus on design smells and the principles of object-oriented application design in the Processing language. Processing is a language that is derived from Java, but omits certain object-oriented concepts of the Java language. Therefore, it allows students to create a simple application in just a few lines of code, but also allows the student to write conceptually working code that omits the rules of good application design. In this study, the following research questions will be answered:

1. What design smells apply to Processing code and which of these occur the most in novice programmers code?
2. What causes novice programmers to write the most commonly seen design smells in Processing code?
3. How could underlying causes of these design smells in Processing code written by novice programmers be detected automatically?

To answer these research questions, a set of design smells that apply to the Processing language will be defined. The
occurrence of these design smells in novices code will be
determined by manually analyzing novices code and find-
ing the design smells in the code. The results and patterns
from this analysis will be used to propose tools for auto-
matic detection of the defined design smells using static
code analysis.

The next two sections of this paper will discuss the back-
ground and work related to the subject. Section 4 will
explain the research method in-depth. Section 5 describes
the design smells and concepts of good application design
in the Processing language. Section 6 discusses the results
of the code analysis and section 7 discusses the implemen-
tation of automated detection tools for the earlier defined
design smells. Section 8 describes the verification of these
tools on code from different sources. The final sections
contain the conclusion and the discussion.

2. BACKGROUND
This section introduces important concepts related to code
analysis and programming education in the context of this
study.

2.1 Processing
Processing is a Java-based programming language that
hides certain language features while adding easy to use
drawing and interaction possibilities. Processing hides
object-oriented language features such as accessibility of
field and methods from the programmer, while giving ac-
cess to a standard library containing functions to draw
interactively on screen. Each program created in Process-
ing has a draw() method which is run in a loop to animate
the drawings on screen.

Processing was created to make programming of interac-
tive graphics easier, since the creators noticed how diffi-
cult it was to create simple interactive programs in com-
mon programming languages such as Java and C++[10].
Because the programs are primarily meant for creating in-
teractive sketches, the programs written in Processing are
also called sketches.

2.2 Code smells
A code smell is defined as a surface indication which usu-
ally corresponds to a deeper problem in the system. This
means that a code smell only identifies a code segment
that most likely has some correspondence with a deeper
problem in the application design. The code smell itself is
merely an indication for possibly badly written code, hence
the name code smell. An experienced programmer would
by seeing the code immediately see that something odd
is happening, without directly knowing the exact cause of
the problem.

Code smells are widely used in programming education to
detect possible problems and to give feedback to novice
programmer code. Existing research has determined the
most important code smells by standards of programming
literature on professional programming[11]. The code smells
found in this study were used for the assessment of novices
code. This resulted in a framework that could be used for
the assessment of novices code in general.

2.3 Design smells
Design smells are structures in the design that indicate a
violation of fundamental design principles and negatively
impact design quality[12]. Although design smells seem to
originate as part of code smells, design smells are usually
more abstract. Whereas code smells are a surface indi-
cation of a deeper problem, design smells imply a deeper
problem with the application design itself. Design smells
are difficult to detect using static code analysis, since de-
sign smells are related to the application as a whole, as
opposed to parts of the code.

2.4 Static code analysis
Static code analysis is the analysis of code without execu-
ting or compiling the code. The analysis is done solely
by interpreting the flow of the source code, instead of us-
ing the bytecode or another form of compiled code. Static
code analysis can be used effectively to detect many kinds
of code smells. It can also, although with less accuracy,
point out some design smells.

Static code analysis is often used in combination with unit
tests for automated feedback tools in programming edu-
cation. Sometimes, only static code analysis is used. A
framework for automated feedback in programming edu-
giving feedback.

A popular automated feedback tool that is based on static
code analysis is PMD. PMD finds code style issues as well
as code smell issues. PMD works for multiple program-
ning languages and custom rules can be implemented in
Java.

3. RELATED WORK
A lot of research has been performed on computer science
education and programming education in general. How-
ever, most existing research has focussed on code smells
and automated feedback tools based on static code analy-
sis and sets of generated tests. Research on design smells
and application design for novice programmers is missing.
This is notable since many novice programmers are having
problems with application design. In 2005, a large survey
was conducted in order to gain a better understanding of
problems that students face when learning to program[6].
This study found that abstract concepts in programming
are not the only difficulty that novice programmers are fac-
ing. Many problems arise when novices have to perform
program construction and have to design the program.
Because of the lack of research on design smells, research
related to code smells is most relevant. Although design
smells are different from code smells, they are closely re-
lated. Most recently, studies were conducted on the oc-
currence of code smells in novice block based languages.
Studies found that code written by novices in languages
such as Microsoft Kodu and LEGO MINDSTORMS EV3
do contain code smells, with the lazy class, duplicate code
and dead code smells occurring the most[5]. Another re-
search on the language Scratch that was conducted on
community code and code of kids new to programming has
also shown comparable results. This study also shows that
the duplicate code and dead code smells are commonly seen
in novices code[1]. Although these studies were focusing
on code smells and conducted on block based languages,
they show that code analysis is important and that code
smells indeed do occur in novices code. Furthermore, the
duplicate code smell is closely related to multiple design
smells.

A closer related study on providing quality feedback has
been performed by T. Blok. This study has provided rules
to improve the feedback of 25 common errors in novices
code using the static analysis tool PMD[2]. This research
has primarily studied the possibilities to give solution-
based feedback on non-working code, this study focuses
on design problems in working code.
4. METHOD OF RESEARCH
Every research question in this study has its own method of research. The answer to each research question is used by the method of the next research question.

The first research question is answered by analyzing Processing code for design smells. The relevant design smells are obtained from two sources. The first source is teachers and teaching assistants in a Processing programming course. The second source is the existing literature on code smells in object-oriented programming languages. The occurrence of the design smells in the code is determined by manual analysis where PMD falls short.

The Processing code that is analyzed for the first research question comes from multiple sources. The first source consists of two batches of code written by novice programmers of the study Creative Technology at the University of Twente. The first batch is written mid-way in the programming course, whilst the second batch is written as the final project for the same course. Both batches are written by the same group of students and all code is provided anonymously. The second source is community written code that is used for comparison to the first source. This code originates from www.openprocessing.org and was retrieved on the 20th of May 2017. The code on this website is selected by picking code pieces with most likes during all time the community has been active. The number of likes on each piece of code determines the popularity of the code, as they are awarded by the community members themselves. Sketches from this source might contain professional code, but also badly written code, as the sketches are selected based on popularity as opposed to quality.

The second research question is answered using analysis of the novices code that actually contains design smells. By discussing different code examples and design smells with a small group of teaching assistants and teachers the causes of each design smell are found. These causes will be used to answer the second research question and will be used to provide feedback when doing automated analysis in the third research question.

To answer the third and final research question, the findings of the first and second research question are combined in order to detect the design smells using static code analysis. The study will propose custom rules for the static analysis tool PMD which should be able to detect the design smells found in the first research question effectively. In order to let PMD work with Processing code, the Processing code will be converted to Java first.

To test if the proposed tools can actually be used to detect the design smells with high accuracy, the tool is applied to the earlier analyzed programs as well as some new sets of programs from different sources. The results of this test answer the third and final research question.

5. DESIGN PRACTICES AND SMELLS
By interviewing teaching assistants and a teacher of the programming course on Processing and by manually analyzing the sets of code from both sources, the eight most occurring design smells as well as rules for good design that apply to Processing were determined. The following sections will discuss the different design smells and also discusses the best design practices for Processing code.

5.1 Discussion on design practices
Since Processing is a fairly new language, no precise rules on application design have been proposed or defined. This study defines some design smells for Processing application based on best practices from other object-oriented programming languages, such as Java. It should, however, be noted that some good practices that apply to many object-oriented languages do not apply in Processing. For example, Processing omits visibility of properties and fields. This causes all fields to be public; they may be written to from different scopes without the use of getters and setters. This is considered bad practice is most languages, but is common and accepted in Processing.

Also, design smells that apply to other languages may not apply to Processing. In Java, event handlers that handle multiple tasks by branching are considered a design smell[8], but in Processing, this is the only way to handle multiple events. Therefore, handling multiple events by branching in one event handler is considered the best way to handle events.

Because of these incompatibilities, defining best practices for application design in Processing is not trivial. The smells defined in this paper are based on experiences with programming in Processing. They might be changed if the best practices of the Processing language change or if new programming practices are proposed.

5.2 Processing related design smells
The following section contains design smells that were defined while discussing code snippets with teachers and teaching assistants. These design smells are specific to Processing and follow from best practises when creating a sketch in Processing.

Pixel hardcode ignorance
The pixel hardcode ignorance smell is a design smell specific to Processing sketches. It refers to having no abstraction for positioning elements that are drawn in the sketch. In the following example, a rectangle and car are drawn on the screen. They are both hardcoded in pixels.

```java
void draw() {
    // Pixels are hardcoded
    rect(30, 40, 10, 20);
    car.draw();
}

class Car {
    // Partial class
    PImage image;

    void draw() {
        // Position is hardcoded in pixels
        car(image, 60, 60);
    }
}
```

In this case, moving, scaling or animating the sketch is difficult and in more involved sketches code duplication will occur. The programmer should use constants and variable to determine the position of drawn object, where possible abstracting from pixels.

Decentralized event handling
The decentralized event handling smell is a Processing specific smell that occurs when a novice programmer uses the global event variables in processing to perform event handling in draw methods, child classes or threads.

Processing defines global variables such as `mouseButton`, `mouseX`, `mouseY`, `key` or `keyCode`. These global variables can be requested from anywhere in the code, but are meant to be used inside the event handling methods, such as `mouseMoved()`, `mousePressed()` or `keyTyped()`. An novice programmer would think of these variables as global and put them in the main draw method, instead of using an event handler for each event. This makes for a harder to understand code.

```java
void draw() {
    mouseMoved();
    mouseClicked();
    keyPressed();
}
```
programmer may actually choose to not use these methods, and use the variables directly from other parts of the code, such as:

```java
void draw() {
    if (keyPressed & key == 'B') {
        color = 0;
    } else {
        color = 255;
    }
}
```

In this example, the `fill(int)` method changes the color of the drawings as soon as the key B is pressed. Although this code will work perfectly, it is smelly, since events can better be handled through the `keyPressed()` method, for example:

```java
int color = 255;
void keyPressed() {
    if (key == 'B') color = 0;
}
```

This code has the same functionality but handles the keyboard event inside the `keyPressed()` method, which is considered more readable and maintainable. Programmers of Processing sketches should always use the methods for event handling instead of putting event variables everywhere in the application.

### Drawing state change

The drawing state change smell refers to changing the state of global variables in the sketch from the draw method loop. In Processing, the `draw()` method runs in a loop to redraw elements on the screen, unless `noLoop()` is used in the `setup()` method. Although the `draw()` method is meant for drawing objects on the screen as part of the sketch, it can be used as any other method, which makes it possible to change the state of the sketch during execution. While this should only be used to animate objects, it is often used for incrementing counters and doing calculations, which should happen in a different place.

A novice programmer may, for example, include some kind of counter, for example:

```java
final int PERIOD = 100;
int counter = 0;
char[] text = {'s', 'w', 'e', 't'};
void draw() {
    int bound = min(text.length, frameCount/PERIOD);
    for (int i = 0; i < bound; i++) {
        text(text[i], 10*i+10, 50);
    }
}
```

This code is not considered smelly since it does not change any of the global variables in the sketch from the `draw()` method.

### Decentralized drawing

The decentralized drawing smell occurs in Processing sketches if draw methods are called in methods that are not part of the call stack of the `draw()` method. All things drawn on the screen should always be drawn in either the `draw()` method itself, or in methods that are (indirectly) called by the `draw()` method. They should not occur in methods like the `setup()` method or the event handling methods. A novice programmer may write the following application, which contains the decentralized drawing design smell because a rectangle is drawn from the `setup()` method.

```java
void setup() {
    rec(10, 20, 30, 40);
}
```

### 5.3 Object-oriented design smells in Processing

#### Stateless class

A stateless class is a class that defines no fields. It only defines methods that get data via parameters. This causes the class to have no internal state: there are no properties defining the condition of the object.

```java
class Stateless {
    void someMethod(Object obj) {
        //Do something with obj.
    }
}
```

In some languages such as Java, this kind of classes are sometimes called utility classes and are perfectly allowed. They help moving out computations and manipulators from stateful classes. This has some benefits, such as the stateless classes being completely immutable and therefore thread-safe[3].

In Processing, stateless classes are considered a design smell. Since Processing allows having global methods in a sketch (which are defined in the hidden parent class of the sketch), utility methods should be defined here. Therefore, stateless classes should not occur in a Processing sketch.

#### Long method

The long method design smell is a smell that is directly related to the method length code smell. When a method exceeds a certain size, the method performs too many actions and should be split or shortened. Methods that have this design smell usually perform multiple algorithms or computations in one method, when they actually should be split into multiple methods.

Based on comments from teaching assistants and the Processing documentation, methods longer than 25 lines of code are considered long methods. They should be split or shortened.

#### Long parameter list

The long parameter list design smell occurs when a method accepts too many parameters. When a method exceeds a
certain amount of parameters, the method either performs too many tasks or a (sub)set of the parameters from a certain amount of parameters, the method either performs too many tasks or a (sub)set of the parameters from an object.  

```java
void drawLine(int x1, int y1,
int x2, int y2,
int red, int green,
int blue) {
    /*
    * In this example, red, green and blue are
    * one color, x1 and y1 are one position, x2
    * and y2 are one position, which should have
    * been only 3 parameters.
    */
}
```

In the first case, the long method design smell will probably occur and the method should be split or shortened. In the second case, a stateful class containing the parameters should be defined which represents the object as a single parameter. Based on functions defined in the Processing documentation, methods that accept 6 or more parameters are considered methods with a long parameters list. They should be altered to accept fewer parameters.

God Class

The God Class smell denotes complex classes that have too much responsibility in an application. It is detected by combining three software metrics: the Weighted Methods Count (WMC), the Access To Foreign Data (ATFD) metric and the Tight Class Cohesion (TCC) metric. The God Class smell is defined more in-depth in the book Object-Oriented Metrics in Practice [7]. In Processing, the parent class of the sketch has a great chance of being a God Class, because programmers have access to all fields and functions defined on the top-level at all times. This can cause child classes to interleave with the parent class which causes the metrics to go bad quickly. A God Class is considered bad design since it reduces maintainability and readability.

6. OCCURRENCES AND CAUSES

In the last section, eight design smells that apply to Processing are discussed. Analysis on code from both sources has been done to determine how much these design smells occur. Also, the likely causes for the design smells are researched. Analysis of the code has shown that design smells occur very frequently in novices code. In the first analyzed set, each program contained one or more code smells. In the second set of code written by novices, 72 of the 79 analyzed programs contained one or more code smells, as noted in Table 1.

Although this might not be surprising since novices are still learning to program, the number of programs that do contain one or more smells in community code might be. 160 out of 178 programs that were analyzed contained one or more code smells. There could be multiple reasons causing this. It could be that the community code is mostly written by inexperienced programmers, but it is more likely that to Processing programmers, the rules for good design are unclear. This is of course caused by Processing being a young language and lacking defined design guidelines.

More interesting to know is which design smells occur the most in the different sets of programs. Table 2 has an overview of the occurrences of each analyzed smell in each set. As we can see, from this overview, the three most occurring smells are the long method, pixel hardcode ignorance and decentralized event handling smells. Also, the drawing state change smell occurs in many community programs, while this smell occurs less inside novices programs. This might be caused by the novices assessment. They are asked as part of the assessment to implement classes, something that community programmers do not necessary have to do.

6.1 Causes

Using manual analysis and by using the input of teachers the causes of each design smell are investigated. This section describes the causes of each smell.

Pixel hardcode ignorance

In novices code, this smell occurs because there is no abstraction from drawing elements having a location as opposed to an object having a location on the screen. Objects still have a static position on the screen directly written as pixels inside the program. A better way to handle this is by giving the object class a position and using this position to draw the structures belonging to the class. In community code, this smell occurs mostly in small programs that draw something simple. The programmer felt no need for abstraction since the sketch is very small.

Decentralized event handling

The decentralized event handling smell occurs for two main reasons. The first reason is the opportunity to decentralize event handling. Because Processing has a lot of global variables, it is really easy to work around the event methods. Many programmers tend to do this because they can easily integrate it in their existing code, losing abstraction from event handling. Another reason for this to happen is drawing on the position of the mouse pointer. This is done in many programs. The best solution is to use the `mouseMoved()` method in combination with a position variable used for drawing.

Drawing state change

The drawing state change smell occurs for different reasons. One of the reasons is the programmer wanting to animate an object by incrementing, decrementing or calculating some value on each redraw. Most of the times, this can be fixed because the calculated value should actually be part of an object, but sometimes, this might not be the case. Pointing to a specific solution is diffi-

<table>
<thead>
<tr>
<th>Set</th>
<th># of programs</th>
<th># with smells</th>
</tr>
</thead>
<tbody>
<tr>
<td>Novices 1 (N1)</td>
<td>61</td>
<td>61</td>
</tr>
<tr>
<td>Novices 2 (N2)</td>
<td>79</td>
<td>77</td>
</tr>
<tr>
<td>Community (C)</td>
<td>178</td>
<td>160</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Smell</th>
<th>N1</th>
<th>N2</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pixel hardcode ignorance</td>
<td>90.2%</td>
<td>86.1%</td>
<td>50.6%</td>
</tr>
<tr>
<td>Decentralized event handling</td>
<td>85.2%</td>
<td>62.0%</td>
<td>32.0%</td>
</tr>
<tr>
<td>Drawing state change</td>
<td>42.6%</td>
<td>20.3%</td>
<td>55.6%</td>
</tr>
<tr>
<td>Decentralized drawing</td>
<td>4.9%</td>
<td>5.1%</td>
<td>3.4%</td>
</tr>
<tr>
<td>Stateless class</td>
<td>9.8%</td>
<td>2.5%</td>
<td>1.7%</td>
</tr>
<tr>
<td>Long method</td>
<td>80.3%</td>
<td>72.2%</td>
<td>36.5%</td>
</tr>
<tr>
<td>Long parameter list</td>
<td>19.7%</td>
<td>7.6%</td>
<td>10.1%</td>
</tr>
<tr>
<td>God Class</td>
<td>16.4%</td>
<td>3.8%</td>
<td>10.7%</td>
</tr>
</tbody>
</table>
cult in those cases. Another reason for this smell is using counters, which can actually always be replaced by the frameCount variable.

**Decentralized drawing**
There is no clear reason for programmers having the decentralized drawing smell. Some novices treat Processing as a sophisticated drawing application and draw from all methods since they do not understand the programming language that Processing is. Also, some novice students may have the misconception that drawing is a way to record a state change. However, drawing should always occur from the draw() method.

**Stateless class**
The stateless class smell is mostly caused by the programmer not understanding the principle of object-oriented programming. The programmer moves out long methods by putting them in separate classes which are used as utility classes, which is considered bad design in Processing, because of the global scope. It is also caused by the programmer failing to grasp a central object-oriented concept, namely that data should be bundled with associated methods operating on that data. In that case the properties for a certain class are read from everywhere, but are not bundled inside the class.

**Long method**
In more than half of the programs, long methods are caused by drawing complex structures that have to be split into different parts or even different objects. This mostly causes the program to have a too long draw() method. This should be fixed by splitting up the draw method. In all other cases, the long method smell is caused by putting all application logic inside one method. This is of course fundamentally bad design and should be changed.

**Long parameter list**
Novice programs that contain the long parameter list smell mostly have this smell because they define methods as a way to combine a set of methods into one. They want to repeatedly draw some structure with slightly different parameters, so they put the small sequence of functions inside a method with a lot of parameters. In most cases, the best way to fix this is by creating an object out of the structure that the programmer wants to draw. In some cases, this is not possible, but the number of arguments should be reduced by combining parameters into objects.

**God class**
The god class smell only occurs in a small set of programs and is caused by the programmer not understanding the responsibility of its defined classes. Each class defined by the programmer has multiple responsibilities or one responsibility is divided over multiple classes. This causes these classes to communicate heavily with the global parent scope, pushing the metrics of the program to bad values. It can be fixed by reconsidering the responsibilities of each class.

7. **IMPLEMENTATION OF AUTOMATED DETECTION**
This section discusses the implementation of rules used for automated detection of the earlier discussed design smells. In order to implement the automated detection for these smells, static code analysis was chosen. The PMD framework is used to implement the rules, which means that each rule can be used in combination with PMD to detect design smells in Processing code.

**7.1 Processing code analysis in PMD**
In order to make it easier to work with Processing code in PMD, the Processing sketches are converted to Java code first, using the processing-java binary. Since PMD already has a grammar and supporting functions for Java, only the rules still need to be implemented. The rules implemented as part of this study detect the design smells inside the resulting Java files.

Converting the Processing code to Java has some important side effects. First of all, the sketch is converted to one Java class with all additional classes inserted as inner classes of this class. This is because the additional classes need access to the Processing standard library, which is defined by the class PApplet in Java. The generated class extends PApplet to have access to all Processing functions. These functions can then be used by the methods, but also by the inner classes.

If a PMD rule is violated, an violation is added to the PMD report. The rules implemented as part of this study detect when a design smell is found and reports them as a violation to PMD.

**7.2 Implementation of design smells**
Each design smell in this study is implemented as one PMD rule. All rules are implemented as an AbstractJavaRule, which means that PMD can execute them on Java files. Each smell has a different implementation discussed in the following sections.

**Pixel hardcore ignorance**
The pixel hardcore ignorance smell is implemented by checking each method invocation expression. When an expression calls a method, this expression is compared against the list of drawing functions in Processing, as defined in Appendix B. A function matches the expression if and only if the name of the method is equal to the method name called by the expression, the number of parameters specified in the expression does match the number of parameters expected by the method, and the scope of the expression does not define another method with the same name and arguments (e.g., the method is not overridden). When the expression matches the method call of a drawing method, then all parameters of the function that are defined in pixels are checked for being a literal value. When the method is called with a literal value for one parameters that is defined in pixels, a violation is created. In that case the program contains the pixel hardcore ignorance design smell.

**Decentralized event handling**
The decentralized event handling smell uses possible call stacks to determine which methods are allowed to use global event variables, as defined in Appendix C.1. The detection algorithm of the smell consists of two steps. First, the rule checks which event methods as defined in Appendix C.2 are defined by the program. Of these methods, all possible method call stacks are evaluated and saved, as long as the methods can only be called from event handling methods. This detection is done by the exclusive call stack algorithm as described in Appendix A.

The second step of the detection algorithm goes over all expressions in the code. If the expression is not defined inside one of the methods saved earlier, it is checked for the usage of global event variables. If an expression uses the global event variables, then a violation is created.


Drawing state change

The drawing state change smell detection algorithm also consists of two steps. In the first step of the algorithm, the algorithm determines all methods that are called as part of the draw sequence. This is done by the non-exclusive call stack algorithm as described in Appendix A. All methods that are part of this sequence are saved for use in step 2. In step 2 of the algorithm, each expression inside the draw sequence is checked for the usage of variables that are defined in the top-level scope (e.g., the main class of the program). If such a variable is used, and the expression is a self-assignment or the variable is used as the left-hand side of an expression, then the variable is mutated, indicating a drawing state change. In that case, a violation is created because the state of the application has changed.

Decentralized drawing

The decentralized drawing smell detection rule is implemented using a similar algorithm as the decentralized event handling smell. In the first step of the algorithm, all methods that are exclusively called as part of the draw sequence are determined. This is done by the exclusive call stack algorithm as described in Appendix A. These methods are saved for usage in step 2. In step 2 of the algorithm, for each expression in the program is checked if it is called by a method that is part of the exclusive call stack as determined in step 1. If the expression is not part of the draw sequence, it is checked if the expression is a method call. When an expression calls a method, this expression is compared against the list of drawing functions, as defined in Appendix B. Like the implementation of the pixel hardcode ignorance detection algorithm, a function matches the expression if and only if the name of the method is equal to the method name called by the expression, the number of parameters specified in the expression does match the number of parameters expected by the method, and the scope of the expression does not define another method with the same name and arguments. When the expression matches the method call of a drawing method, then a violation is created.

Stateless class

The stateless class smell detection rule is implemented by going over all class and interface definitions. When the definition is an inner class, not an interface, and not defined abstract, then the fields declared in the class are checked. If the class does not declare any fields, then a violation is created and the class is considered stateless. Please note that the algorithm only runs on inner classes, which are in Processing just the classes that are defined by the programmer. The top level class which declares the main program is not checked, since in the Processing language, this is not seen as a class.

Long method

The long method smell detection rule is implemented using the same algorithm as PMD uses to check the method count of Java classes. The rule uses the NCSS (Non Commenting Source Statements) algorithm to determine just the lines of code in the method. When this exceeds 25, a violation is created.

Long parameter list

The long parameter list smell detection rule is implemented using the same algorithm as PMD’s ExcessiveParameterList rule. For each method definition, the amount of accepting parameters is counted. If this count exceeds 5, then a violation is created.

God Class

The God Class smell detection rule is reimplemented based on the rule that was provided by PMD to detect the God class in Java files. A shortcoming of this algorithm is that it calculates the needed metrics, the Weighted Methods Count (WMC), the Access To Foreign Data (ATFD) metric and the Tight Class Cohesion (TCC) metric, one time for the whole compilation unit. That means the rule does not take into account inner classes as different classes. This makes sense for Java programs, in which inner classes should not be used for defining new standalone objects. In Processing, however, all classes are in the end inner classes of the main program class. Therefore, these classes should be seen as different objects and have their own calculated software metrics.

The new implementation calculates the WMC, ATFD and TCC metrics for each inner class separately. If one of the inner classes violates these metrics, then this class is considered a God class, as opposed to the whole file being a God class. Then for this class, a violation is added.

7.3 Design limitations

The usage of PMD as static code analysis framework introduces some design limitations to the detection of design smells. This section discusses these limitations.

An important limitation of PMD is the call stack detection. To determine which methods are called from a certain method, PMD makes use of the name of the method and the number of arguments that the method is called with. Because PMD has very little knowledge about the type of each variable, it cannot distinguish between different overloaded methods. Also, if a method is called on an object, PMD might not always be able to detect the type of the object the method is called on, which causes the method detection to fail. This limitation affects the rules that actually try to detect method calls. The pixel hardcode ignorance smell might not always report the right overloaded method in the violation, for example. This is however not of great consequence. The feedback is not entirely correct, but the smell detection is. In the decentralized event handling and decentralized drawing rule, this limitation might lead to false positives, since it was impossible to detect the entire event handling stack or draw sequence respectively. For the drawing state change smell, it might lead to false negatives because it was impossible to detect the entire draw sequence.

Another limitation in the proposed rules is the handling of object constructors. Since constructors are handled differently than method definitions in PMD, not all rules will work correctly on them. Constructors will, for example, never be detected as part of the event handling stack or draw sequence. This means the decentralized event handling and decentralized drawing rule will always report violations when using global event variables or drawing methods inside constructors. For the same reason, the change of program variables from a constructor will never be detected as part of the event handling stack or draw sequence. This means the decentralized event handling and decentralized drawing rule will always report violations when using global event variables or drawing methods inside constructors. For the same reason, the change of program variables from a constructor will never be detected as part of the event handling stack or draw sequence. Although these limitations are there, it is expected that the detection will work fine on most of the programs. This will be validated in the next section. 

8. VALIDATION OF AUTOMATED DETECTION

To assure that the proposed PMD rules can indeed detect design smells in Processing applications, they were validated by two tests. In the first test, the correctness of
Table 3. Frequency of occurrences of false positives and false negatives in the sketches of each set.

<table>
<thead>
<tr>
<th>Set</th>
<th>#</th>
<th>FP</th>
<th>FN</th>
<th>FP &amp; FN</th>
</tr>
</thead>
<tbody>
<tr>
<td>N1</td>
<td>61</td>
<td>2</td>
<td>3.3%</td>
<td>11</td>
</tr>
<tr>
<td>N2</td>
<td>79</td>
<td>2</td>
<td>2.5%</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>178</td>
<td>22</td>
<td>12.4%</td>
<td>14</td>
</tr>
<tr>
<td>Total</td>
<td>318</td>
<td>26</td>
<td>8.2%</td>
<td>28</td>
</tr>
</tbody>
</table>

The rules are checked by validating the implemented rules against the results found during manual analysis of the programs. In the second test, the rules are checked for applicability by running the rules on a new set of programs. The tests will be further discussed in the next sections.

8.1 Correctness

To assure correctness, the proposed PMD rules were executed on the first and second novices set (N1 and N2 respectively), as well as the community set (C). The results of the execution were compared against those of manual analysis to find false positive and false negative detections. In case the PMD rules detected a smell that was not detected during manual analysis, there is one false positive detection counted. If the PMD detection did not detect a smell that was actually found during manual analysis, it counts as a false negative detection.

Table 3 gives an overview of the sketches that have one or more false positives (FP) in the detection, the sketches that have one or more false negatives (FN) in the detection and the sketches that have one or more false detections, either false positives or false negatives.

The results show that in the first novices set and community set, the number of sketches with at least one false detection is about 20%, while it is about 5% for the second novices set. This means the correctness rate for sketches scanned with the PMD rules is about 75%. However, this only shows the results for all rules on each sketch. More detailed results are possible, which show the correctness rate per rule.

In Table 4 the results of the correctness test are given per smell. From this table, it is easy to see that certain smells are more difficult to detect when compared to some other rules. The long parameter list, long method, and stateless class smells are easy to detect because they are just counting rules. It is fairly easy to count lines of code or count the number of defined parameters for a method. In the same way, counting the number of variables defined for a class is fairly simple. More difficult are the Processing specific smells. Still, the pixel hardcode ignorance and drawing state change smells do not contain any false detections, but the decentralized drawing and event handling smells do. This is mostly because the possible call stacks are detected wrong. This is not trivial to do in static code analysis, but might still be improved. The God Class smell gives a lot of false negatives because the rule implementation changes, as discussed in section 7.2. This causes PMD to detect god classes differently from what was used during manual analysis.

All things considered, the results are satisfying. Especially when compared to other different static code analysis tools, as done by [9], the amount of false positives and false negatives can be considered low. The high results are also possible because assumptions can be made based on Processing code. For example, converting Processing code to Java will make sure that all classes of a sketch are part of the same file, which means that all code definitions can be detected inside that file. These assumptions are used in the PMD rules to make the analysis better.

8.2 Applicability

To assure the rules can be applied to a broader set of programs than the ones used in the manual analysis performed earlier, the PMD rules were executed on some new sets of programs that have different behaviour. In this test, a small new set of novices programs is used (N3), as well as some code examples from a programming course on Processing (E) and code examples from the website learningprocessing.com (L). Table 5 shows the frequency of smells in the different sets as detected by the PMD rules.

From this table, it is clear that the code examples and code from learningprocessing.com do not contain as many smells as the previous sets. However, it still seems striking that sketches from this source contain this many code smells. This is because both sets also contain examples of ‘messy’ code, which is effectively code that is meant to be improved by the programmer.

Code from the third novices set is in line with the earlier sets. All sketches in this set contain one or more design smells.

In Table 5 the results are split by smell. This table shows each of the PMD rules as proposed by the study works on an untested set of programs. Only the God class smell was not present in the new sets. This is mostly because the sketches in the set are somewhat smaller than the sets used earlier.

To make sure the rules worked correctly, a sample of 30 sketches were taken from the set. In these 30 sketches, there was 1 false positive and no false negatives. Therefore the tool is considered to be able to detect design smells in different kinds of Processing sketches.

9. CONCLUSIONS

In this paper, a set of design smells for the Processing language has been introduced, as well as rules for automated detection of these design smells using static code analysis. The paper shows that there exist design smells that apply to Processing code, as well as that these smells can be automatically detected with good accuracy.

The paper shows that there exist new design smells that are specific to Processing, such as the pixel hardcode ignorance, decentralized event handling, drawing state change and decentralized drawing smells defined by this paper. Besides these Processing specific design smells, there are also classical object-oriented design smells that apply to Processing code, such as the stateless class, long method, long parameter list and God Class smells. These were already defined earlier for different programming languages, but also apply to Processing. The paper also shows that the pixel hardcode ignorance, decentralized event handling and long method smells occur the most in novices code.

Furthermore, the paper concluded that there are different causes for the occurrence of each design smell. Many of these causes are related to a bad understanding of application structures or design in general.

Finally, the paper proposed detection rules implemented using the PMD static code analysis tool to automatically detect design smells in Processing code. These proposed rules were checked against manually analyzed sets of Processing sketches as well as new code to test the applicability of the rules to a broad set of sketches. The results show that the proposed way of detecting design smells per-
forms well on the code examples used in this study. Some rules might still need improvement, but the basic design for each rule has been introduced.

10. DISCUSSION AND FUTURE WORK

This study has introduced a selected set of design smells that apply to Processing. In the future, more research on design smells in Processing will be needed to determine which design smells are most important and to determine if formal design guidelines are needed. Furthermore, the set of design smells selected by this paper are based on design smells in novices code from one source, which is the programming course for Creative Technology on the University of Twente. It might be the case that different programming courses score very different on the number of design smells in the programs.

Moreover, this paper only investigates the causes of each design smell by means of interviewing a small group of teaching assistants and teachers. Therefore, the causes of each design smell are not researched in detail. Future research should investigate the causes of design smells more thoroughly to be able to understand why the design smells occur in the Processing language.

Future research could also work on improving the smell detection rules as proposed by this paper. The rules as proposed by this paper are not optimized and it may be possible to detect design smells with an even higher accuracy, especially the code smells that solely apply to Processing sketches. Another point of interest may be un-explored design smells. It might be that there are more design smells that apply to the Processing language.

11. REFERENCES


APPENDIX

A. PSEUDOCODE CALL STACK DETECTION

This section describes 2 algorithms that are used for the call stack detection. Algorithm 1 describes the algorithm used for detecting the call stack of a method, whilst Algorithm 2 determines which methods of these call stack are only called as part of this call stack.

function callStack(method);
Input : The method to determine the possible call stacks of
Output: Call stack as set of methods
create a set of methods s and add method;
i = 0;
while i ≠ the size of the method set do
  i = the size of the method set;
  foreach method m in s do
    foreach method c in methods called by m do
      if c not in s then
        add c to s;
      end
    end
  end
end
Algorithm 1: Determine call stack of a method

function exclusiveCallStack(method);
Input : The method to determine the exclusive call stack for
Output: Set of methods that are only called as part of the call stack of method
s = callStack(method);
foreach method m in s do
  foreach method c in methods that call m do
    if c not in s then
      remove m from s;
    end
  end
end
Algorithm 2: Determine call stack of a method

B. LIST OF DRAWING FUNCTIONS

This section lists the known drawing functions in Processing. Only methods that directly draw on screen are listed. float types are listed as f to make the definition shorter. A parameter listed with an asterix (*) means the parameter should be provided in pixels, which is used for the pixel hardcode ignorance smell detection.

- arc(f*, f*, f*, f*)
- arc(f*, f*, f*, f*, f, f)
- ellipse(f*, f*, f*, f*)
- line(f*, f*, f*, f*)
- line(f*, f*, f*, f*, f*, f*)
- point(f*, f*)
- point(f*, f*, f*)
- quad(f*, f*, f*, f*, f*, f*, f*)
- rect(f*, f*, f*, f*)
- rect(f*, f*, f*, f, f)
- rect(f*, f*, f*, f, f, f)
- triangle(f*, f*, f*, f*, f, f*, f, f*)
- bezier(f*, f*, f*, f*, f*, f*, f*, f*, f*)
- bezier(f*, f*, f*, f*, f*, f*, f*, f*, f*, f*, f*)
- bezierPoint(f*, f*, f*, f*, f)
- bezierTangent(f*, f*, f*, f*, f)
- curve(f*, f*, f*, f*, f*, f*, f*, f*)
- curve(f*, f*, f*, f*, f*, f*, f*, f*, f*, f*)
- curvePoint(f*, f*, f*, f*, f)
- curveTangent(f*, f*, f*, f*, f)
- box(f*)
- box(f*, f*, f*)
- sphere(f*)
- shape(PShape)
- shape(PShape, f*, f*)
- shape(PImage, f*, f*)
- shape(PImage, f*, f*, f*, f*)
- shape(char, f*, f*)
- shape(char, f*, f*, f*)
- shape(String, f*, f*)
- shape(char[], int, int, f*, f*)
- shape(String, f*, f*, f*)
- shape(char[], int, int, f*, f*, f*)
- shape(String, f*, f*, f*, f*)
- shape(f, f*, f*)
- shape(f, f*, f*, f*)
- shape(int, f*, f*)
- shape(int, f*, f*, f*)

C. LIST OF EVENT METHODS AND GLOBALS

C.1 Event globals

The following variables are global event variables in Processing: mouseButton, mousePressed, mouseX, mouseY, pmouseX, pmouseY, key, keyCode and keyPressed.

C.2 Event methods

The following methods are called by Processing as event handling methods. Please note that usage of the event parameters is optional, the method might also be defined without any parameters:

- mouseClicked(MouseEvent)
- mouseDragged(MouseEvent)
- mouseMoved(MouseEvent)
- mousePressed(MouseEvent)
- mouseReleased(MouseEvent)
- mouseWheel(MouseEvent)
- keyPressed(KeyEvent)
- keyReleased(KeyEvent)
- keyTyped(KeyEvent)