The Smell of PROCESSING

Remco de Man and Ansgar Fehnker
Faculty of Electrical Engineering, Mathematics, and Computer Science, University of Twente, Enschede, The Netherlands
r.j.p.deman@student.utwente.nl, ansgar.fehnker@utwente.nl

Keywords: Software Smells, Code Smells, Design Smells, Programming Education, Software Design.

Abstract: Most novice programmers write code that contains design smells which indicates that they are not understanding and applying important design concepts. This is especially true for students in degrees where programming, and by extension software design, is only a small part of the curriculum.
This paper studies design smells in PROCESSING a language for new media and visual arts derived from *Java*. Language features – as well as common practices in the PROCESSING community – lead to language specific design smells. This paper defines design smells for PROCESSING, informed by a manual analysis of student code and community code.
The paper describes how to detect these smells with static analysis. This serves two purposes, first to standardize design requirements, and second to assist educators with giving quality feedback. To validate its effectiveness we apply the tool to student code, community code, and code examples used by textbooks and instructors. This analysis also gives a good sense of common design problems in PROCESSING, their prevalence in novice code, and the quality of resources that students use for reference.

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. Within these degrees, programming, and by extension software development, is just one subject among others. The curriculum will offer only limited room to teach principles and practices of software application design.

For novices in these fields programming is often difficult. Many novices will be able to write code that effectively solves a given problem, but have trouble with applying principles of good application design. As soon as they have to build bigger applications, they face unnecessary complexities of their own making. These make it difficult for students to understand, maintain or extend their own or other students 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(Hermans and Aivaloglou, 2016).

The field of software development has developed in the last decades a set of principles and practices that guide good application design. Their aim is to make code understandable, maintainable, and extendable. A *design smell* is an indicator of poor design (Suryanarayana et al., 2014) that may negatively affect these aims.

In programs by novice programmers these smells are often symptoms of not understanding design principles or how to put them in practice. Giving quality feedback on application design to these programmers is especially important, and justifies the need for automated tools that assists programming educators.

This paper studies design smells in PROCESSING code. The definition of these smells is informed by a manual analysis of PROCESSING code from two different sources: student code, and community code. The paper then describes how automated static analysis tools can be used to detect these smells. The tool is then applied to code from three sources: student code, community code, and code examples used by textbooks and instructors. This analysis gives a good sense of common design problems in PROCESSING, their prevalence in novice code, and the quality of resources that students use for reference.

The PROCESSING language is a derivative of *Java* for electronic art, new media, and visual design (Reas and Fry, 2010). It is used in the context of a first year component of the degree *Creative Technology* at the *University of Twente*. This degree has no degree-specific entry requirements and attracts a very diverse

Man, R. and Fehnker, A.
The Smell of PROCESSING.
In *Proceedings of the 10th International Conference on Computer Supported Education (CSEDU 2018) - Volume 2*, pages 420-431
ISBN: 978-989-758-291-2

student population; students with extensive to no prior experience with programming and application design.

The PROCESSING language was derived from *Java* with an emphasis on ease of use, with the explicit intention to be used for teaching programming. It offers built-in methods for 2D and 3D graphics and for handling user input events, and a host of libraries for graphics, audio and video processing, and hardware I/O.

PROCESSING is a subset of *Java*, but omits certain object-oriented concepts of the *Java* language. These simplifications allow students to create a simple application in just a few lines of code, but still allows students to write working code that ignores rules of good application design.

While PROCESSING is a subset of *Java*, design smells that have been developed for Java do not always translate to PROCESSING. One reason is that PROCESSING consciously omits certain features, such as a system of access control. In PROCESSING all fields are `public` by design, which by itself would be considered a design smell in *Java*. Another reason is the established practice of the PROCESSING community. It is not uncommon in PROCESSING programs for objects to access and change fields of other objects directly. This would be considered poor practice in *Java*.

However, just because the language does not explicitly enforce certain coding practices, does not mean that students should not be exposed to the ideas behind them. The software development community has learned that certain practices lead to poor code, regardless of language particularities. We have to strike a balance between keeping the benefits of a simplified language, while introducing well-known object-oriented design principles. While it is technically possible to use any *Java* keyword or syntax in PROCESSING, and teach "*Java* by stealth", we choose instead to teach PROCESSING as it is defined by its inventors, and introduce design in the form of design smells that are to be avoided.

This paper will introduce customisations of existing design smells to PROCESSING, and in addition, design smells that are particular to PROCESSING. All PROCESSING programs have a shared basic structure, with the same predefined methods for event handling and status variables. This common structure makes them suspectable to common violations of good design principles, violations you usually will not find in *Java* programs.

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. Static analysis tools that detect those design smells are then a complementary tool to define and standardize these guidelines.

In the recent years, a lot of research has been performed on automated feedback frameworks for students using static code analysis, which primarily look at styling issues and possible bugs. Although successful in providing feedback on these aspects of the code, few give feedback on the application design. Feedback generated by industrial software development tool is mostly based on software metrics, such as cyclomatic complexity. These concepts are, not surprisingly, poorly understood by novice programmers. They will not understand the feedback, nor will it help them to gain a deeper understanding of object-oriented application design.

This study will focus on design smells and the principles of object-oriented application design in the PROCESSING language. 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, this paper defines a set of design smells that apply to the PROCESSING language. It reports on the results an manual analysis of these design smells in novice's code. The results and patterns from this analysis will inform how we extend a static analysis tool, PMD, for automatic detection of the defined design smells .

The next section discusses the background and work related to the subject. Section 3 will explain the research method. Section 4 describes the design smells and concepts of good application design in the PROCESSING language. Section 5 discusses the results of the manual code analysis and section 6 discusses the implementation of automated detection tools for the earlier defined design smells. Section 7 validates the effectiveness these tools on code from different sources. The final sections contain the conclusion and the discussion.

2 BACKGROUND AND RELATED WORK

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

PROCESSING. PROCESSING was created to make programming of interactive graphics easier, since the creators noticed how difficult it was to create simple interactive programs in common programming languages such as *Java* and *C++* (Reas and Fry, 2010). It based on *Java*, but hides certain language features such as access control of field and methods, while providing a standard library for drawing interactively on screen. Each program created in PROCESSING has a `draw()` method which is run in a loop to animate the drawings on screen. Because the PROCESSING programs are primarily meant for creating interactive sketches, they are also called *sketches*.

Code Smells. A code smell is defined as a *surface indication* which usually 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 suspect that something odd is happening, without always directly knowing the exact cause of the problem.

Code smells are widely used in programming education to give feedback to novice programmer code. Existing research has determined the most important code smells defined by standard literature on professional programming(Stegeman et al., 2014). The code smells found in this study were used for the assessment of novice's code. This resulted in a framework that could be used for the assessment of novice's code in general. This framework covered nine criteria for code quality, of which two are related to design: decomposition and modularization.

Most recently, studies were conducted on the occurrence of code smells in block based languages such as *Microsoft Kodu* and *LEGO MINDSTORMS EV3*. In (Hermans et al., 2016) is is reported that *lazy class*, *duplicate code* and *dead code* smells occurr the most. Another study on *Scratch* code from the community and on code of children new to programming showed comparable results (Aivaloglou and Hermans, 2016). The *duplicate code* smell is closely related to multiple design smells.

Design Smells. Design smells are structures in the design that indicate a violation of fundamental design principles and negatively impact design quality(Suryanarayana et al., 2014). Although design smells seem to originate as part of code smells, design smells are usually more abstract. Design smells imply a deeper problem with the application design itself. Design smells are difficult to detect using static code analysis, since design smells are related to the application as a whole, as opposed to parts of the code.

There is only little research on design issues. In 2005, a large survey was conducted in order to gain a better understanding of problems that students face when learning to program (Lahtinen et al., 2005). This study found that abstract concepts in programming are not the only difficulty that novice programmers are facing. Many problems arise when novices have to perform program construction and have to design the program.

Static Code Analysis. *Static code analysis* analyzes code without executing or compiling the code. Static analysis is used in industry to find problems with the structure, semantics, or style in software. This includes simple errors, like violations of programming style, or uninitialized variables, to serious and often difficult to detect errors, such as memory leaks, race conditions, or security vulnerabilities.

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 programming languages and custom rules can be implemented in *Java*. PMD was used in (Blok and Fehnker, 2016) to investigate to what extent PMD covers 25 common errors in novice's code. It furthermore linked the common errors to misunderstood concepts, in order to give students appropriate feedback on the root cause of the error.

FrenchPress uses static analysis to provide feedback to students, however it is explicitly not aimed at novice programmers, nor does it cover design problems (Blau, 2015).

Keunig et al. reviewed 69 tools for providing feedback to programming exercises (Keuning et al., 2016). They found that the majority of tools use testing based techniques, and most often they point to programming mistakes, which could be called code smells. They found only one tool that uses static analysis to explain misunderstood concepts. The tool, CourseMarker (Higgins et al., 2005), offers a range of tools, among them also a tool to analyse object-oriented diagrams for design issues.

3 METHOD OF RESEARCH

Each research question in this study has its own method. The first question is what design smells apply to PROCESSING code and to what extend they occur in novice programmers code. The relevant design smells are from two sources: (1) teachers and teaching assistants in a PROCESSING programming course, and (2) literature on code smells in object-oriented programming languages.

The occurrence of the design smells in actual code is determined by manual and semi-manual analysis of programs from two sources. The first source consists of two batches of code written by novice programmers of the degree *Creative Technology* at the *University of Twente*. The first batch of 61 programs was written for the final tutorial in the programming course, whilst the second batch of 79 programs was 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. This batch of 178 programs originates from `www.openprocessing.org` and was retrieved on the 20^{th} of May 2017. We selected the most popular code, i.e. programs that received most likes by community members since the community has been active. Sketches from this source might contain professional code, but also poorly written code. This source is of particular interest to us, as students will use the examples that they find on this site for inspiration for their own project.

The second research question relates to the causes of the most common design smells in PROCESSING code. Using examples from novice's code that actually contains design smells, we discussed possible causes with a small group of teaching assistants and teachers. The results of this discussion will informed the development of automated analysis in the third research question.

The third and final research question is if these design smells can be detected with static analysis. We 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.

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 new sets of programs. This contains a third set of student code, and code from teaching resources: examples provided by teaching staff and code that accompanies the textbook. For each of these we determine the false positive rate as well as the false negative rate.

4 DESIGN SMELLS FOR PROCESSING

This section presents eight design smells as well as rules for good design that apply to PROCESSING. Four of these are specific to PROCESSING. They are a consequence of the predefined basic structure of PROCESSING programs, with the predefined methods for event handling and status variables. This lead to smells that will not appear in common *Java* programs.

The other four design smells are based on existing design smells in *Java*. While PROCESSING is a subset of Java, design smells that have been developed for Java do not always translate directly. One reason is that PROCESSING consciously omits certain features, such as a system of access control. In PROCESSING all fields are public by design, which by itself would be considered a design smell in *Java*. A another reason is the predefined basic structure of all PROCESSING program, with its predefined methods and status variables for event handling. In *Java* event handlers that handle multiple tasks by branching are considered a design smell(Lelli et al., 2017), but in PROCESSING, this is the only way to handle multiple events.

4.1 PROCESSING Specific Design Smells

These design smells are specific to PROCESSING and relate to best practices when creating a sketch in PROCESSING.

Pixel Hardcode Ignorance. The pixel hardcode ignorance smell refers to having no abstraction for positioning elements that are drawn in the sketch. Instead of modelling objects with a position or size that can be represented on screen they treat PROCESSING as advanced drawing tool for rectangles and ellipses.

In the following example, a rectangle and car are drawn, both hardcoded in pixels.

```
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. It will be difficult to impossible to reuse the code in a different context, or turn it into a proper object.

This smell is related to "magic numbers", the use of literals instead of variables and constants. This is considered a code smell in other languages, but both community code as well as standard textbooks on PROCESSING use magic numbers liberally. This is in part simply because PROCESSING does not use the concept of user defined constants. Failure to abstract from the position of a graphical element, however will often prevent them from producing working, extendable or maintainable animations. This elevates this smell to a design smell. Use of magic number for other purposes, such as margins or sizes is accepted practice.

Jack-in-the-box Event Handling. The Jack-in-the-box event handling smell, a form of decentralised event handling, occurs when a novice programmer uses the global event variables in processing to perform event handling outside of dedicated event handling methods. PROCESSING defines global variables such as `mouseX`, `mouseY`, `mouseButton`, `key` or `keyPressed`. These global variables can be requested from anywhere in the code, but are meant to be used inside the event handling methods, such as `keyTyped()`, `mouseMoved()`, or `mousePressed()`.

A novice programmer may actually choose to not use these methods, and use the variables directly from other parts of the code, such as:

```
void draw() {
  if (keyPressed && key == 'B') {
    fill(0);
  } else {
    fill(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, as follows:

```
int color = 255;

void keyPressed() {
  if (key == 'B') color = 0;
}

void keyReleased() {
  if (key == 'B') color = 255;
}

void draw() {
  fill(color);
}
```

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.

This smell often causes students to struggle with debugging, as it becomes very difficult to trace changes on screen to events, and vice versa. The name of the smells refers to the surprise many students or teachers feel, when they find in some remote part of the program code that unexpectedly handles events. And often is the cause of intricate bugs.

Drawing State Change. 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 calculations and updates. These which should happen in a different place, preferably in methods that belong to an object and update its state.

Decentralized Drawing. The decentralized drawing smell occurs in PROCESSING sketches if drawing 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. You might find that the event handler will directly draw something on screen, instead of changing the state of an object, which then changes the representation of the object.

4.2 Object-Oriented Design Smells in PROCESSING

The smell mentioned in this section are known smells of languages such a *Java*, and are also common in PROCESSING code. However, they may have to be adapted to fit with the particularities and practices of PROCESSING.

Table 1: Result of manual analysis for the three different sets of programs.

Set	Number of programs	Lines of code per program	Smells per program	Smells per 1000 lines	Programs with some smell
Novices (tutorial)	61	310	3.5	11.3	100.0%
Novices (finals)	79	154	2.6	16.8	97.5%
Community code	178	162	2.0	12.3	89.9%

Stateless Class. A stateless class is a class that defines no fields. It only defines methods that get data via parameters. In *Java* classes of this kind 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(Goetz, 2006).

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. Stateless classes should rarely or never 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.

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 actually should be abstracted as part of an object.

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* (Lanza, 2006).

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.

5 DESIGN SMELLS IN CODE

In the previous section, eight design smells that apply to PROCESSING are discussed. Analysis on code from both sources has been done to determine how often these design smells occur. This section will also discuss the likely causes for the design smells.

5.1 Manual Analysis

Table 1 shows the results of the manual analysis. The code written by novices for the final tutorial has about twice as many lines of code as the code written for the final project, or by the community. This is in part due to the fact that the students were working towards the final tutorial over four weeks, and slowly building up the program, and in part due to the fact that they had limited experience in using object-oriented concepts to organise code.

Design smells occur very frequent in novice's code written for the tutorial. All programs contained at least one design smell, and on average 3.5 of the 8 smells we considered. The second set of code, written by novices for their final project, did improve somewhat. They contained on average only 2.6 smells, and only 72 of the 79 analyzed programs contained one or more code smells.

The results for novices may not be surprising since novices are still learning how to program. Surprising is however, the number of community programs that do contain one or more smells. 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 also likely that to PROCESSING programmers the rules for good design are unclear or less important. This is of course caused by PROCESSING being a language without the history and broad usage of other languages, which led in those languages to widely accepted programming guidelines. Within PROCESSING the focus often lies on quickly producing visually appealing prototypes,

Table 2: Percentages of programs in different sets that exhibit a given design smell.

Smell	*Novice (tutorial)*	*Novice (finals)*	*Community code*
Pixel hardcode ignorance	90.2%	86.1%	50.6%
Jack-in-the-box event handling	85.2%	62.0%	32.0%
Drawing state change	42.6%	20.3%	55.6%
Decentralized drawing	4.9%	5.1%	3.4%
Stateless class	9.8%	2.5%	1.7%
Long method	80.3%	72.2%	36.5%
Long parameter list	19.7%	7.6%	10.1%
God Class	16.4%	3.8%	10.7%

instead of on building software systems that will have to be extended and maintained.

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 Jack-in-the-box 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 necessarily have to do.

5.2 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 novice's 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.

Jack-in-the-box Event Handling. The Jack-in-the-box 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 for event handling, it is really easy to work around the event methods. Many programmers tend to do this because they can quickly integrate it in their existing code, losing abstraction from event handling. That can lead to code that is littered with small bits of 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, however the easiest method, and the first students learn, is to use the global variables `mouseX` and `mouseY`.

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 difficult in those cases. Another reason for this smell is using counters, which can actually almost 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 that PROCESSING is a fully fledged programming language, instead of a simple scripting language. 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. 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. Instead variables in the main class are read from everywhere, and are not encapsulated inside of an appropriate 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.

6 AUTOMATED DETECTION

This section discusses the implementation of rules used for automated detection of the earlier discussed design smells. 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.

6.1 PROCESSING Code Analysis in PMD

In order to make analysis of PROCESSING code with PMD possible, the PROCESSING sketches are converted to *Java* code, 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. For example, 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, a 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.

6.2 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 Hardcode Ignorance. The pixel hardcode 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. 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 parameter that is defined in pixels, a violation is created. In that case the program contains the pixel hardcode ignorance design smell.

Jack-in-the-Box Event Handling. The Jack-in-the-box event handling smell uses possible call stacks to determine which methods are allowed to use global event variables. This smell required to implement extend PMD with an detection algorithm. The detection algorithm of the smell consists of two steps.

First, the rule checks which of the predefined PROCESSING event methods are implemented and used 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 as described in (de Man, 2017).

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 (de Man, 2017). 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 Jack-in-the-box 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 (de Man, 2017). These methods are saved for usage in step 2.

In step 2 of the algorithm, for each expression in the program it 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 predefined drawing functions of PROCESSING. 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, i.e. 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 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 reported.

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

God Class. The God Class smell detection rule is re-implemented 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.

Table 3: Frequency of false positives and false negatives per design smell in each set.

Smell	Novice (tutorial)				Novice (finals)				Community code				Total			
	FP		FN		FP		FN		FP		FN		FP		FN	
Pixel hardcode ignorance	0	0%	0	0%	0	0%	0	0%	0	0%	0	0%	0	0%	0	0%
Jack-in-the-box event handling	0	0%	1	1.6%	0	0%	0	0%	0	0%	1	0.6%	0	0%	2	0.6%
Drawing state change	0	0%	0	0%	0	0%	0	0%	0	0%	0	0%	0	0%	0	0%
Decentralized drawing	2	3.3%	0	0%	2	2.5%	0	0%	22	12.4%	0	0%	26	8.2%	0	0%
Stateless class	0	0%	0	0%	0	0%	0	0%	0	0%	0	0%	0	0%	0	0%
Long method	0	0%	0	0%	0	0%	0	0%	0	0%	0	0%	0	0%	0	0%
Long parameter list	0	0%	0	0%	0	0%	0	0%	0	0%	0	0%	0	0%	0	0%
God Class	0	0%	10	16.4%	0	0%	3	3.8%	0	0%	13	7.3%	0	0%	26	8.2%
Total	2	3.3%	11	18.0%	2	2.5%	3	3.8%	22	12.4%	14	7.9%	26	8.2%	28	8.8%

6.3 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 Jack-in-the-box 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 Jack-in-the-box 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 not cause the drawing state change rule to detect a violation.

Despite these limitations, it is expected that the detection will work fine on most of the programs. This will be validated in the next section.

7 VALIDATION

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 the rules is 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.

7.1 Correctness

To assure correctness, the proposed PMD rules were executed on the two novices sets, as well as the community set. 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 the results of the correctness test 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. The pixel hardcode ignorance and drawing state change smells do not contain any false positives or negatives, while the decentralized drawing

Table 4: Results for the automated analysis, as checked by the proposed PMD rules.

Set	Number of programs	Lines of code per program	Smells per program	Smells per 1000 lines	Programs with some smell
Novices (resit)	17	297	3.1	10.5	100.0%
Course material	32	81	0.8	10.4	43.8%
Textbook examples	149	40	0.9	23.0	54.4%

and event handling smells do. This is mostly because the possible call stacks are detected incorrectly. Our current analysis uses a straightforward algorithm that can still be improved, as discussed in the previous section. The God Class smell gives a lot of false negatives because the rule implementation changed, as discussed in Section 6.2. This causes PMD to detect god classes differently from the criteria used during manual analysis.

All things considered, the results are satisfying. Especially when compared to the state of the art in static code analysis tools, as reported in (Okun et al., 2013). A rate of 8.2% and 8.8% of false positives and false negatives, respectively, can be considered to be low. Our analysis is helped because we can make certain assumptions about PROCESSING code. For example, converting PROCESSING code to *Java* will make sure that all classes of a sketch are part of one file, which means that all code definitions can be detected inside that file. These assumptions can be exploited in the PMD rules to improve the analysis.

7.2 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 three new sets of programs that have different behaviour. The first set is a new set of novices programs written for the same final project, as were the programs from the set that we considered before. The difference is that these are submissions for a resit. Students had to take the resit most commonly because their initial submission was found to be lacking. The second set are code examples from our first year programming course that uses PROCESSING; examples provided by lecturers and assistants. The third set of code examples are taken the website `learningprocessing.com`. They are the example accompanying the first 10 chapter of the textbook *Learning Processing*. This is the content that is covered in the course.

Table 4 shows the results for the different sets as detected by the PMD rules. It is apparent that novice's code differs significantly from the course and textbook example, not just because it contains many more lines of code. The code examples for the course and code from the textbook do not contain as many smells as the novice sets. However, it still seems striking that sketches from this source contain this many code smells. This is in part because both sets also contain examples of 'messy' code, which is effectively code that is meant to be improved by the student. It also includes examples from the first weeks that illustrate basic concepts, before more advanced concepts are taught. For example, canonical PROCESSING examples on the difference between global and local variables will exhibit the draw the state change, since it is a very visual illustration of the difference. However, also the course material and textbook examples contain also smells that should be arguably improved.

Table 5: Percentages of programs in different sets that exhibit a given design smell. Compare with Table 2.

Smell	*Novice (resit)*	*Course material*	*Textbook examples*
Pixel hardcode ignorance	70.6%	34.4%	36.3%
Jack-in-the-box event handling	76.5%	21.9%	18,1%
Drawing state change	41.2%	9.4%	21.5%
Decentralized drawing	0.0%	6.3%	13.4%
Stateless class	29.4%	0.0%	0.0%
Long method	82.4%	12.5%	2.7%
Long parameter list	11.8%	0.0%	0.0%
God Class	0.0%	0.0%	0.0%

Code from the third novices set for the resit of the final project is also different from the code for the final project (Table 1). They have more smells and they are significantly longer. By these measurements this code is more similar to the novice code that was written for the tutorial, earlier in the course. This align well with the fact that students that take the resit are behind on the subject.

Table 5 splits the results by smell. This table shows each of the PMD rules on an untested set of programs. Only the God class smell was not present in the new sets. This is of course also in part because the code in the course and textbook set are significantly smaller than the programs in the novice sets.

Interesting is that the *state-less class* smell occurs much more frequent in programs submitted for the resit, than in any other set. Some students were told that they have to use classes to structure the code; they introduced stateless classes, to make a, somewhat misguided, effort towards this request.

To make sure the rules worked correctly, a sam-

ple of 30 sketches were taken from the set. In these 30 sketches, there was 1 false positive and no false negatives. This confirms that the tool is be able to detect design smells in different kinds of PROCESSING sketches.

8 CONCLUSIONS AND FUTURE WORK

This paper applied the concept of design smells to PROCESSING. The new design smells that we introduced relate to common practice by novice programmers, as well as the PROCESSING community. In addition it identified relevant object-oriented smells, that also apply to PROCESSING. We showed the relevance of these new and existing smells to PROCESSING code, by manual analysis of novice code and code by the PROCESSING community. We found that a majority of programs by novices and by the community contain at least some PROCESSING related design smell. We found that these are caused by poor understanding of application design in general, or lack of attention to design.

For the eight identified design smells, we implemented customised checks in PMD. These proposed rules were checked against the manually analyzed sets of PROCESSING sketches to estimate the false positive and false negative rate. They were then applied to a new set of code to demonstrate their wide applicability. The results show that the proposed way of detecting design smells performs well on the code examples used in this study. This analysis also revealed that even course material and textbook examples exhibit, to a somewhat surprising extend, design smells.

This work produced along the way also the first static analysis tool for PROCESSING. It created an automated pipeline, defined new rules, and customized existing rules, all to accommodate PROCESSING specific requirements.

This study has introduced a selected set of design smells that apply to PROCESSING. In the future, more research on design smells will be needed to further develop design guidelines for PROCESSING. Design smells are used in software development practice to guide refactoring of code. Similarly, we need refactoring techniques for PROCESSING code, including a benchmark of well structured programs. This should be accompanied by a review of existing teaching resources, to avoid that unnecessary smells set a poor example.

This paper presents a tool for automated detection, and discusses its accuracy and applicability. Future research has to investigate the most effective use of these tools; whether students should use them directly, or only teaching assistants, to help them with providing feedback, how frequently to use them, and if and how to intergrade them into peer review, assessment, or grading.

REFERENCES

Aivaloglou, E. and Hermans, F. (2016). How kids code and how we know: An exploratory study on the scratch repository. In *ICER '16*, New York, NY, USA. ACM.

Blau, H. (2015). Frenchpress gives students automated feedback on java program flaws (abstract only). In *SIGCSE '15*, New York, NY, USA. ACM.

Blok, T. and Fehnker, A. (2016). Automated program analysis for novice programmers. In *HEAd17*. Universitat Politecnica de Valencia.

de Man, R. (2017). The smell of poor design. In *26th Twente Student Conference on IT*. University of Twente.

Goetz, B. (2006). *Java concurrency in practice*. Addison-Wesley, Upper Saddle River, NJ.

Hermans, F. and Aivaloglou, E. (2016). Do code smells hamper novice programming? A controlled experiment on scratch programs. In *ICPC 2016*, pages 1–10.

Hermans, F., Stolee, K. T., and Hoepelman, D. (2016). Smells in block-based programming languages. In *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*.

Higgins, C. A., Gray, G., Symeonidis, P., and Tsintsifas, A. (2005). Automated assessment and experiences of teaching programming. *J. Educ. Resour. Comput.*, 5(3).

Keuning, H., Jeuring, J., and Heeren, B. (2016). Towards a systematic review of automated feedback generation for programming exercises. In *ITiCSE '16*, New York, NY, USA. ACM.

Lahtinen, E., Ala-Mutka, K., and Järvinen, H.-M. (2005). A study of the difficulties of novice programmers. *SIGCSE Bull.*, 37(3).

Lanza, M. (2006). *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer, Berlin New York.

Lelli, V., Blouin, A., Baudry, B., Coulon, F., and Beaudoux, O. (2017). Automatic detection of GUI design smells: The case of blob listener. *arXiv preprint arXiv:1703.08803*.

Okun, V., Delaitre, A., and Black, P. E. (2013). Report on the static analysis tool exposition (SATE) IV. *NIST Special Publication*, 500:297.

Reas, C. and Fry, B. (2010). *Getting Started with Processing*. Maker Media, Inc.

Stegeman, M., Barendsen, E., and Smetsers, S. (2014). Towards an empirically validated model for assessment of code quality. In *Koli Calling '14*, pages 99–108, New York, NY, USA. ACM.

Suryanarayana, G., Samarthyam, G., and Sharma, T. (2014). *Refactoring for Software Design Smells: Managing Technical Debt*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition.