Detecting and Addressing Design Smells in Novice PROCESSING Programs

Ansgar Fehnker(✉) and Remco de Man

Formal Methods and Tools Group, Faculty of Electrical Engineering, Mathematics and Computer Science, University Twente, Enschede, The Netherlands
ansgar.fehnker@utwente.nl

Abstract. Many novice programmers are able to write code that solves a given problem, but they struggled to write code that adheres to basic principles of good application design. Their programs will contain several design smells which indicate a lack of understanding of how to structure code. This applies in particular to degrees in which programming, and by extension software design, is only a small part of the curriculum.

This paper defines design smells for PROCESSING, a language for new media and visual arts that is based on *Java*. This includes language specific smells that arise from the common structure that all PROCESSING programs share. The paper also describes how to detect those smells automatically with static analysis. This tool is meant to support teaching staff with providing feedback to novices on program design.

We applied the tool to a large set of student programs, as well as programs from the PROCESSING community, and code examples used by textbooks and instructors. The latter gave a good sense of the quality of resources that students use for reference. We found that a surprising number of resources contains at least some design smell. The paper then describes how to refactor the code to avoid these smells. These guidelines are meant to be practical and fitting the concepts and constructs that are known to first-year students.

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.

This paper is based on the experience in the first year of *Creative Technology* (CreaTE) at the University of Twente. CreaTe equips students with skills and knowledge required to develop creative and innovative human-centered applications. Programming is an integral part of the curriculum. First-year students are introduced to the PROCESSING language, a recent derivative of Java for electronic

B. M. McLaren et al. (Eds.): CSEDU 2018, CCIS 1022, pp. 507–531, 2019.
https://doi.org/10.1007/978-3-030-21151-6_24

art, new media, and visual design. It is a good fit for first-year CreaTe students, as its philosophy – as exemplified by a thriving community – is very close to the design philosophy of CreaTe. It prepares the ground for teaching more generic languages, such as Java and C++, as well as for teaching languages used in physical computing, such as the Arduino sketch language.

For novices, programming is often difficult. At the end of the first-year, students 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 find it difficult to understand, maintain or extend their own or other students code. Lack of exposure to principles of software design frustrates good students, who cannot progress beyond a certain point, and hinders struggling students, who are facing unnecessary complexities of their own making. 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 [1].

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 [2] 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 into practice. Giving quality feedback on application design to these programmers is especially important, and justifies the need for automated tools that assist programming educators.

This paper studies design smells in Processing, a language developed by Casey Reas and Ben Fry in the 2000s for visual arts and new media [3]. Processing lowers the bar to producing interactive applications and 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. It is a subset of *Java*, and omits, for example, certain object-oriented concepts of the *Java* language. These simplifications allow students to create a simple application in just a few lines of code.

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. Processing is only an introductory language, and CreaTe students will proceed to common general-purpose languages like Java, C# and C++. At that

point, prior exposure to essential design concepts will be beneficial to understand advanced language features in those languages.

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.

The definition of these smells in this paper is informed by a manual analysis of Processing code from two different sources: student code, and community code. This paper will introduce customisations of existing design smells to Processing, and in addition, define 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.

The paper then describes how automated static analysis tools can be used to detect these smells. The tool is 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.

This paper is an extended version of the work presented in [4]. We refined the definition of some of the Processing specific smells such that they fit better with the accepted practice and extended the prior work with a discussion of recommended refactorings.

The contributions of this paper are as follows:

1. Defining design smells for Processing code and assess to what extent these occur the most in novices as well as publicly available code.
2. A static analysis tool that detects those smells automatically.
3. A discussion of proposed refactorings that will address these smells and improve the overall design of the programs.

The next section discusses the background and work related to the subject. Section 3 describes the design smells, Sect. 4 the results of the manual code analysis. Section 5 discusses the implementation of automated detection tools for the earlier defined design smells. Section 6 validates the effectiveness these tools on code from different sources. In Sect. 7 we discuss possible refactoring that address the smells. The final section contains the conclusion and the discussion.

2 Background and Related Work

2.1 Processing

The Processing language 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++* [3]. It is 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 the screen. Because the PROCESSING programs are primarily meant for creating interactive sketches, they are also called *sketches*.

2.2 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 [5]. 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 [6] it was reported that *lazy class*, *duplicate code* and *dead code* smells occur the most. Another study on *Scratch* code from the community and on code of children new to programming showed comparable results [7]. The *duplicate code* smell is closely related to multiple design smells.

2.3 Design Smells

Design smells are structures in the design that indicate a violation of fundamental design principles and negatively impact design quality [2]. 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 more 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 programming education. In 2005, a large survey was conducted in order to gain a better understanding of problems that students face when learning to program [8]. 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.

2.4 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 [9] 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.

Keunig et al. reviewed 69 tools for providing feedback on programming exercises [10]. 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 [11], offers a range of tools, among them also a tool to analyse object-oriented diagrams for design issues.

This paper is an extended version of the work presented at the *10th International Conference on Computer Supported Education (CSEDU)* [4]. This extended paper uses a refined definition of the Processing specific smells and presents updated results. It elaborates, furthermore, in detail on how to refactor code to achieve a better overall structure.

3 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, which leads 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*. 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 [12], but in Processing, this is the only way to handle multiple events.

3.1 Code Bases

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 79 programs was written for the final tutorial in the programming course, whilst the second batch of 61 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.

3.2 Processing Specific Design Smells

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

3.3 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 an 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.

3.4 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 the 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.

3.5 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.

3.6 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 the screen, instead of changing the state of an object, which then changes the representation of the object.

3.7 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.

3.8 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 [13].

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.

3.9 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.

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

Set	Number of programs	LoC per program	Smells per program	Smells per 1000 LoC	Programs with some smell
Novices (tutorial)	79	154.4	2.0	13.1	88.6%
Novices (finals)	61	310.3	3.1	10.0	98.4%
Community code	178	162.7	1.9	11.4	86.0%

3.10 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.

3.11 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* [14].

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.

4 Design Smells in Processing Code

In the previous section, eight design smells that apply to Processing are discussed. Analysis of code from publicly available sources as well as student code has been done to determine how often these design smells actually occur. Table 1 shows the results of the manual analysis. They show the how many smells are

present in a program. The results take to account whether a smell is present, not how often. Programs often make the same mistake consistently; one novice program, for example, had 106 instances of the *pixel hardcode ignorance* smell, i.e. 106 graphics commands with hardcoded pixels. Here we count this as one program exhibiting the *pixel hardcode ignorance* smell.

The results show that programs for the final project are about twice as large as programs written for the tutorial, with 154.4 lines of code against 310.3 in the finals. In the tutorial code, we find on average 2.0 of the eight considered smells, in the final project 3.1. This is in part due to the fact that the final projects are larger. The number of smells per line code decreases slightly in the code written for the final project. Overall smells occur very frequently in novice code, with close to 88.6% of all tutorial code, and even 98.4% of final projects containing at least one smell.

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

Smell	*Novice (tutorial)*	*Novice (finals)*	*Community code*
Pixel hardcode ignorance	29.1%	49.2%	33.7%
Jack-int-the-box event handling	62.0%	85.2%	32.0%
Drawing state change	20.3%	42.6%	55.6%
Decentralized drawing	5.1%	6.6%	5.6%
Stateless class	2.5%	9.8%	1.7%
God class	3.8%	16.4%	10.7%
Long method	72.2%	80.3%	36.5%
Long parameter list	7.6%	19.7%	10.1%

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

It is 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 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.

5.1 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.

5.2 Design Smell Detection

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.

5.3 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 that define the position 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.

This rule has been slightly modified since [4], to align with the common practice in PROCESSING to define the size of graphical elements as literal values. The definition used in this paper considers only whether the position of an element is hardcoded, while it will not warn if, for example, the width or height is. Fehnker et al. discussed in [15] three dimensions for assessing static analysis warnings: severity, incidence, and correctness. While the definition in [4] was technically correct, and pointed to a definite violation (in contrast to a potential violation), it found many warnings that PROCESSING programmers will not consider severe enough to change the code. In this paper, we use a definition of the smell that will not warn about these less severe instances of *Pixel Hardcode Ignorance*.

5.4 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 extend PMD with a 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 [16].

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.

5.5 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 [16]. 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.

5.6 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 [16]. These methods are saved for use 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.

5.7 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.

5.8 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.

5.9 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.

5.10 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.

5.11 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 of 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.

Table 3. 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	297.7	2.8	9.3	100.0%
Textbook examples	149	40.1	0.8	19.8	51.7%
Course material	31	82.1	0.5	5.9	29.0%

6 Validation

To assure that the proposed PMD rules can indeed detect design smells in Processing applications, we consider two criteria. The first is if they are capable to detect smells on a new set of programs. The second is the false positive rate of the warnings.

6.1 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 contains 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 from the website `learningprocessing.com`. They are the example accompanying the first 10 chapter of the textbook *Learning Processing* [17]. These are the chapters that are covered in the course.

Table 3 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 these sources contain this many code smells.

The high number of smells is explained 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 *Drawing state change* smell, since drawing the state change is a very visual illustration of the difference. However, the course material and textbook examples contain also smells that should be improved. We will discuss some possible refactorings in Sect. 7.

Table 4. Percentages of programs in different sets that exhibit a given design smell. Compare with Table 2.

Smell	*Novice (resit)*	*Textbook examples*	*Course material*
Pixel hardcode ignorance	35.3%	23.5%	6.5%
Jack-int-the-box event handling	76.5%	18.1%	19.4%
Drawing state change	41.2%	21.5%	6.5%
Decentralized drawing	0.0%	13.4%	6.5%
Stateless class	29.4%	0.0%	0.0%
God class	0.0%	0.0%	0.0%
Long method	82.4%	2.7%	9.7%
Long parameter list	11.8%	0.0%	0.0%

Table 4 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 *Stateless 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.

6.2 False Positives

The false positive rate tells how many of the warnings were incorrect. This rate is important because it determines the value that users will attach to a warning. For this analysis, we consider a technical definition of correctness, as defined in [15]. For example, the course material contains a program with a *Long Method* smell. The method in questions has 27 instead of the specified maximum of 25 lines of code. While the warning is technically correct – the method is too long – splitting the method into two part would feel artificial, as it draws one single, slightly more complicated graphical element. The warning is not a false positive and will be counted as a correct warning, even though an individual developer may have good reasons not to act upon it.

The results in Table 5 show that there was one false positive for *Pixel Hardcode Ignorance*. This student program used `pushMatrix` and `popMatrix` in combination with `translate` and `rotate` to move and rotate a graphical element. In Processing there is a practice to use `pushMatrix` and `popMatrix` as a pair to define local blocks of code where the coordinate system is manipulated. Within these local blocks the position can be considered to be relative.

However, semantically, `pushMatrix` refers to the global coordinate system stack. It is just good practice to always use `pushMatrix` and `popMatrix` pairs to make the effect local. The false positive arose because the student used `pushMatrix` and `popMatrix` separated from the actual drawing. Closer inspection of the control flow showed that it was overall correct, despite the warning.

To analyse this correctly would mean to have a full semantic model of coordinate transformations, something that would exceed the capabilities of PMD.

The false positives found for the *Decentralised Drawing* smell are caused by an idiom that caused the call stack to be incorrect. This idiom occurred in particular in community code, which accounts for 18 of the 21 false positives. The false positive rate for this checks seems high, unfortunately, addressing this would require to modify the presentation of the call stack, as provided by PMD, which is outside of the scope of this paper.

Table 5. Frequency of false positives per design smell.

	Warnings	*False positives*	
Pixel hardcode ignorance	157	1	0.6%
Jack-int-the-box event handling	202	0	0.0%
Drawing state change	183	0	0.0%
Decentralized drawing	61	21	34.4%
Stateless class	16	0	0.0%
God class	6	0	0.0%
Long method	192	0	0.0%
Long parameter list	38	0	0.0%
Total	855	22	2.6%

Not counted as false positives were 3 instances of the *Decentralised Event Handling* smell, that pointed to dead code. Technically the warning is correct, as the code contains drawing instructions that are not part of call stack of the main `draw` routine, simply because this code is not part of any call stack. Different programmers may disagree whether this dead code is a problem that needs to be addressed.

From this table, it is easy to see that the long parameter list, long method, and stateless class smells are easy to detect. After all, 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.

All things considered, the results are satisfying. Especially when compared to the state of the art in static code analysis tools, as reported in [18], a rate of 2.6% of false positives 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 Refactorings

The previous section defined the design smells and discussed how to detect them. This section will discuss the causes as well make suggestions how to refactor the code. The refactorings are meant to be practical and suitable for novice programmers. This, for example, excludes patterns in the spirit of the Gang-of-Four [19], since novice programmers are not yet familiar with the required concepts to competently implement those patterns.

7.1 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. Students often view PROCESSING as a type of scripting language for a drawing application, instead of a fully fledged programming language. Graphical objects still have a static position on the screen directly written as pixels inside the program.

A typical example would be the following part of a program that is meant to display parts of a *hamburger*.

```
stroke (0);
fill(#E80000);
rect(263, 254, 60, 8);
rect(200, 254, 60, 8);
rect(137, 254, 60, 8);
```

These line of code fail to capture that they are conceptually related to a graphical object, a hamburger, and more importantly, that the position is determined by the position of the hamburger.

A basic refactoring would be to introduce position variables. For simple programs that consist only of the main class these would most likely be global variables, such that they can be updated by other methods. The exercise to introduce position variables will also clarify which drawing instruction belongs to which graphical object in the animation. In the following example, all three rectangles are part of the hamburger.

```
stroke (0);
fill(#E80000);
rect(hamburgerX+63, hamburgerY, 60, 8);
rect(hamburgerX,    hamburgerY, 60, 8);
rect(hamburgerX-63, hamburgerY, 60, 8);
```

This prepares the ground for the next step, namely to determine which parts deserve their own class. Once classes are introduced, global variables that relate to the hamburger will become part of the `Hamburger` class. In the course we use the following rule of thumb: if you can legitimately ask "Where is the hamburger?", then the hamburger deserves variables modelling the position.

A more sophisticated refactoring uses `pushMatrix` and `popMatrix`. The command `pushMatrix` pushes the matrix representing current coordinate system

onto a stack. The program can then use transformations and rotations to achieve the desired result. Calling `popMatrix` will then restore the previous coordinate system afterwards. The following would be the corresponding refactoring of the hamburger code snippet:

```
stroke (0);
fill(#E80000);
pushMatrix();
translate(hamburgerX,hamburgerY);
rect(63, 0, 60, 8);
rect(0,  0, 60, 8);
rect(-63,0, 60, 8);
popMatrix();
```

Note, that in this code, the first two arguments of the call to `rect` are not absolute positions, even though they are literals. They are relative to the new origin after translation to the position of the hamburger. The width and height however are literals. In other languages novices would usually be encouraged to avoid such magic numbers, and be asked to introduce constants instead. However, textbooks on Processing do not introduce constants; it is not a official feature of the language. It is possible to use the `final` keyword from Java, however this is then strictly speaking no longer Processing. In Processing it is accepted and normal to use literals for dimensions of graphical elements.

It is important to note that `pushMatrix` and `popMatrix` work on a global stack. This means that there is no syntactic requirement to use them in pairs. However, it is established practice to have each `pushMatrix` matched by a `popMatrix` in the same block. The single false positive for the *Pixel Hard-code Ignorance* smell was caused by a program that separated `pushMatrix`, `popMatrix`, and the drawing into different methods.

7.2 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 a global counter on each redraw. This is particularly the case for programs in the very first weeks, that only contain the main class. In this case, it is usually possible to replace the global variable by the predefined `frameCount` variable. In most other cases this can be fixed because the calculated value should actually be part of an object. This means the global variable should become a field, and a method to update the object should take care of manipulating its value.

7.3 Jack-in-the-box Event Handling and Decentralized Drawing

We will deal with these together because they are other two aspects of the same problem: not distinguishing between event handling, drawing, and updates of the state.

Main Processing Tab	Class A	Class B
`global variables` Use for information concerning the entire update, and variables for important objects.	`attributes` What's important to know about an object of this class	`attributes` What's important to know about an object of this class
`setup` Initialize the global variables, set size, create global objects	`constructor` Initializes the attributes for a new object	`constructor` Initializes the attributes for a new object
`draw` All code that drives the display of objects, All code that drives the update	`display methods` One or more methods to display the different parts	`display methods` One or more methods to display the different parts
`event handlers` This are predefined methods such as mousePressed or keyPressed	`update methods` These methods will update attributes as time (frames) go by.	`update methods` These methods will update attributes as time (frames) go by.
`utility methods` Useful methods that don't belong to any particular class.	`event handlers` Update attributes in response to events.	`event handlers` Update attributes in response to events.

Fig. 1. Suggested structure for simple interactive PROCESSING applications.

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 multitude of global variables for event handling, it is tempting to work around the event methods. This usually leads 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, which is done in many programs. The best solution is to use instead the `mouseMoved`() method in combination with a position variable used for drawing. Unfortunately, the easiest method, and the first students see in community code and teaching material is to use the global variables `mouseX` and `mouseY` directly and everywhere where it is convenient.

The *Decentralized Drawing* smell is happening for less obvious reasons. Some novices treat PROCESSING as a sophisticated drawing application, and do not yet understand that PROCESSING is a fully fledged programming language. Also, some novice students have the misconception that drawing is a way to record a state change.

To address these two smells we recommend distinguishing in the code between the three main task of a PROCESSING program: (1) Displaying a graphical object, (2) updating the state of these object from frame to frame, (3) and handling events. Figure 1 gives a recommended structure that will be applicable to most interactive applications that are developed by novices.

Each interesting graphical object in the application should be modelled as a class that encapsulates the relevant state and provides the methods to display, update, and handle events that relate to object. The rule of thumb we give to students is that "If you see a monster on the screen, we want to see a monster class in the project."

The structure in Fig. 1 entails that drawing should ideally happen in draw methods of the corresponding object. These should be only called from the main draw method, or draw methods of other objects. Separate from these should be update-methods that are executed with every frame of the animation. These methods update the state in response to the passing of time. These have to be called from the main draw method, or from other update methods.

Separate from draw- and update-methods should be event handling methods, which should be called from the event handling methods in the main class, such as `mouseMoved()`, or from other event handling methods. This ensures that it is possible to track the calls from the main class to the relevant event handler, which help during debugging. The event handling methods are ideally also the only methods that use global event variables such as `keyCode` or `keyPressed`.

The structure given in Fig. 1 distinguishes between the model, the display, and event handling, not unlike the Model-View-Controller pattern. However, here we separate these horizontally within a class, instead of vertically into separate classes. Our structure is closer to the structure that is also present in *openFrameworks* [20], a toolkit for interactive application based in C++. Note also that this structure uses composition of objects as the main mechanism to compose systems, instead of inheritance. In the first year of CreaTe inheritance is only covered cursory; a more thorough treatment of inheritance takes place when languages such as Java, or C++ are introduced.

7.4 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. Refactoring means to determine whether these methods belong to an object or are general purpose methods. In the latter case, it is usually best to keep them in the main class of the PROCESSING application.

7.5 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. In such cases it may be better to divide the code into smaller methods, however, as mentioned in Sect. 6.2, this depends on the case. In most 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.

7.6 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.

7.7 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.

7.8 Application to Course Material

We applied these recommended refactorings to the course material that we developed ourselves. The main problem was *Jack-in-the-box Event Handling*. It was in most cases easy to move the problematic code out to the predefined event handling methods. This has two added advantages; one is that it gives more attention this often neglected feature of Processing; the other is that it also makes the code more readable. For example, instead of an object chasing the mouse, it is now converging to an explicit `target`, and the `target` is changed by moving the mouse. It makes the relation between the target and the mouse explicit. Changing the code to having other events change the aim is now easily accomplished. Other programs that were refactored contained the *Long Method* and *Pixel Hardcode Ignorance* smell.

The 31 refactored programs now contain two "messy" programs that contain combined 6 smells. These are intentionally poor programs that students have to improve. It contains one program that explains the difference between value and reference passing, which intentionally uses drawing to illustrate the difference, and an example that illustrate an array algorithm, by manipulating and drawing the content of a global array. Finally, it still contains one other example with a long method as discussed before. It was decided not split this method, also to convey that smells are just indicators of potential problems instead of mandatory guidelines. The number of programs with smells reduced from 9 to 5. This includes the two messy programs. Instead of 0.5 smells per program (Table 3), the examples now contain only 0.3 smells on average; or 0.15 if we take disregard the two intentionally "messy" programs.

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, we identified and adapted

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. This is particularly true for the newly proposed Processing specific smells.

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 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 extent, design smells.

This led us to discuss suggested refactoring that will improve the overall design of the application, and reduce the number of smells. We applied these refactoring to the course material that we developed for the course.

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. This paper made a start with discussing some potential refactorings, and by applying them to our own course material. As such this made a first effort towards the future work discussed in [4], which this paper extends. This paper refined the definition of some of the Processing specific smells such that they match more closely the accepted practice, and extended the prior work with a discussion of recommended refactorings. It will take more effort to review other publicly available material and to initiate and contribute to the discussion on the importance of application design within the Processing community.

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.

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, 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. The concept of design smell may prove to be easier for novices to grasp.

The sources and analysed programs that were used in this paper will be available at http://wwwhome.ewi.utwente.nl/~fehnkera/smell/.

References

1. Hermans, F., Aivaloglou, E.: Do code smells hamper novice programming? A controlled experiment on scratch programs. In: ICPC 2016, pp. 1–10 (2016)
2. Suryanarayana, G., Samarthyam, G., Sharma, T.: Refactoring for Software Design Smells: Managing Technical Debt, 1st edn. Morgan Kaufmann Publishers Inc., San Francisco (2014)
3. Reas, C., Fry, B.: Processing: A Programming Handbook for Visual Designers and Artists. The MIT Press, Cambridge (2007)
4. Man, R., Fehnker, A.: The smell of processing. In: Proceedings of the 10th International Conference on Computer Supported Education, vol. 2 (2018)
5. Stegeman, M., Barendsen, E., Smetsers, S.: Towards an empirically validated model for assessment of code quality. In: Koli Calling 2014, pp. 99–108. ACM, New York (2014)
6. Hermans, F., Stolee, K.T., Hoepelman, D.: Smells in block-based programming languages. In: 2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC) (2016)
7. Aivaloglou, E., Hermans, F.: How kids code and how we know: an exploratory study on the scratch repository. In: ICER 2016. ACM, New York (2016)
8. Lahtinen, E., Ala-Mutka, K., Järvinen, H.M.: A study of the difficulties of novice programmers. SIGCSE Bull. **37**, 14–18 (2005)
9. Blok, T., Fehnker, A.: Automated program analysis for novice programmers. In: HEAd 2017, Universitat Politecnica de Valencia (2016)
10. Keuning, H., Jeuring, J., Heeren, B.: Towards a systematic review of automated feedback generation for programming exercises. In: ITiCSE 2016. ACM, New York (2016)
11. Higgins, C.A., Gray, G., Symeonidis, P., Tsintsifas, A.: Automated assessment and experiences of teaching programming. J. Educ. Resour. Comput. **5**, 5 (2005)
12. Lelli, V., Blouin, A., Baudry, B., Coulon, F., Beaudoux, O.: Automatic detection of GUI design smells: the case of Blob listener. In: Proceedings of the 8th ACM SIGCHI Symposium on Engineering Interactive Computing Systems, EICS 2016. ACM (2016)
13. Goetz, B.: Java Concurrency in Practice. Addison-Wesley, Upper Saddle River (2006)
14. Lanza, M.: Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems. Springer, Heidelberg (2006). https://doi.org/10.1007/3-540-39538-5
15. Fehnker, A., Huuck, R., Seefried, S., Tapp, M.: Fade to grey: tuning static program analysis. Electron. Notes Theor. Comput. Sci. **266**, 17–32 (2010)
16. de Man, R.: The smell of poor design. In: 26th Twente Student Conference on IT, University of Twente (2017)

17. Shiffman, D.: Learning Processing: A Beginner's Guide to Programming Images, Animation, and Interaction, 2nd edn. Morgan Kaufmann Publishers Inc., San Francisco (2016)
18. Okun, V., Delaitre, A., Black, P.E.: Report on the static analysis tool exposition (SATE) IV. NIST Spec. Publ. **500**, 297 (2013)
19. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley Longman Publishing Co., Inc., Boston (1995)
20. Perevalov, D., Tatarnikov, I.S.: openFrameworks Essentials. Packt Publishing, Birmingham (2015)