Code Smells in Education Programming Languages

Case studies on LEGO MINDSTORMS EV3 and Kodu

Felienne Hermans
Delft University of Technology
Mekelweg 4
Delft, the Netherlands
f.f.j.hermans@tudelft.nl

Kathryn T. Stolee
North Carolina State University
Raleigh, NC, USA
ktstolee@ncsu.edu

David Hoepelman
Delft University of Technology
Mekelweg 4
Delft, the Netherlands
D.J.Hoepelman@student.tudelft.nl

***Abstract*—Millions of people in the workforce today write code, without degrees or professional training in software development. These end-user programmers perform a variety of tasks, from combining web information to building models that support business decisions. Software engineering research into code smells has traditionally focused on professionally-used object-oriented programming languages, yet these end-user domains and languages also suffer from code smells.**

In this work, we explore the hypothesis that end-user smells also occur in languages aimed at programming education. First, we distill a catalog of end-user smells from existing work on object-oriented smells plus their applications to two end-user domains: Excel and Yahoo! Pipes. Second, we explore the occurrence of end-user smells by examining two end-user languages not previously targeted by smell detection and refactoring research, both aimed at education: LEGO MINDSTORMS EV3 and Microsoft's Kodu. The results of this application show that object-oriented-inspired smells indeed occur in educational end-user languages and are present in 88% and 93% of the EV3 and Kodu programs, respectively. Most commonly we find that programs are plagued with lazy class, duplication, and dead code smells, with duplication smells being present in nearly two thirds of programs in both languages. We further propose new end-user smells inspired by the educational languages, moving beyond the object-oriented paradigm.

I. Introduction

End-user programmers are said to outnumber professional programmers three times over [1]. These end-user programmers perform a wide variety of tasks within their organizations, ranging from creating new web streams to building and maintaining applications in a spreadsheet. When performing these tasks, end-user programmers face many of the challenges of professional developers, such as identifying faults, debugging, or understanding code written by someone else [2].

Similar to professional development is the longevity of the produced artifacts; the average lifespan of a corporate spreadsheet, for example, is five years [3]. During this long lifespan, end-user artifacts are modified, often by different people. These properties make them, like source code artifacts, vulnerable to *smells*.

Code smells, as outlined in the taxonomy of Fowler [4], pertained to object-oriented (OO) code and professional programming languages and were the focus for at least the first decade of code smell and refactoring research [5]. Recently, however, smells in end-user programming have also received attention in research, most notable structural smells in Yahoo! Pipes web mashups [6] and Excel spreadsheets [7]. Experiments in these and other end-user areas have shown that end-user programmers understand smells, and often prefer versions of their code that are non-smelly [8]–[10].

In this paper we broaden research on end-user code smells by examining the domain of educational programming languages. Smells are particularly interesting in educational languages as the programs are often shared and remixed through online communities (e.g., Scratch[1], LEGO MINDSTORMS EV3[2], and Microsoft's Kodu[3] all have online repositories for sharing). We gathered 44 programs written by children and online community members in two languages, EV3 and Kodu, and studied the occurrences of code smells.

The results of this evaluation show that OO-inspired smells in fact occur in both end-user education languages: 88% of the EV3 and 93% of the Kodu programs contained at least one smell. This underlines the power of the code smells concept: they are applicable to visual languages aimed at education, which are quite different from the textual languages aimed at professional developers for which the concept was initially designed.

In EV3 and Kodu, the smells that we most commonly find are small abstractions (lazy class), duplication, and dead code, illustrating commonality across all end-user programming domains. The contributions of this work are as follows:

- Definition of end-user programming smells in LEGO MINDSTORMS EV3 and Kodu (Section III)
- Two case studies investigating end-user smells in educational programming languages: LEGO MINDSTORMS EV3 and Kodu (Section IV and Section V)
- Identification of future opportunities for domain-specific, non-OO-inspired smell detection in end-user programming domains (Section VI)

II. Background

In this work, we use smells previously explored in end-user programming languages to guide our exploration and analysis of programs written in the education languages. In this

[1]https://scratch.mit.edu/studios/11807/

[2]http://www.lego.com/en-us/mindstorms/gallery

[3]Games can be shared within the application or on Xbox Live.

TABLE I
OVERVIEW OF CODE SMELLS PREVIOUSLY STUDIED IN END-USER PROGRAMS

OO Smell	Excel	Yahoo! Pipes
Dead Code		Unnecessary Module [9]
Deprecated Interface		Deprecated Module or Invalid Source [9]
Duplicate Code	Duplicated Formulas [8]	Duplicate Modules, Duplicate String or Isomorphic Paths [9]
Feature Envy	Feature Envy [7]	
Inappropriate Intimacy	Inappropriate Intimacy [7]	
Lazy class or Middle Man	Middle Man [7]	Unnecessary Abstraction [9]
Long Method	Multiple Operations [8]	Noisy Module : Duplicate Field [9]
Many Parameters	Multiple References [8]	
Message Chain	Long Calculation Chain [8]	
No-op		Unnecessary Module [9]
Unused Field		Noisy Module : Empty Field [9]

section, we summarize previous work on code smells in end-user languages and furthermore we introduce the education languages EV3 and Kodu.

A. End-User Smells

In previous research, code smells, originating from the OO domain, were applied to various other, end-user programming paradigms. Most prominently, research has focused on Yahoo! Pipes, a web mashup language and environment with which RSS feed information could[4] be collected and combined from various sources, and spreadsheets.

Research into end-user language smells has taken two approaches, which are not mutually exclusive. The first approach is to take existing smells for OO programming languages, usually those defined by Fowler [4], and transform them to be applicable to the end-user environment [6]–[10]. The second approach is to define smells tailored to the end-user environment. This can be done by interviewing experienced end-users to see which smells they perceive [10], looking at user reports like forum or newsgroup posts [10], [11], or analyzing publicly available repositories [6], [8], [9].

Table I presents an overview of different OO-inspired smells that researchers have found to be applicable to end-user artifacts, as identified in prior work [6]–[9]. Since the smell names for the end-user languages often differ from the OO smell names, we use the OO names throughout the paper. Overall, we often observe few similarities in the code smells studied. Of the 11 smells covered between the two languages, only the *duplicate code*, *lazy class*, and *long method* smells were studied in both, illustrating the breadth of possible code smells, even in smaller, end-user languages.

B. Education Languages

Many languages have been developed with computer science education in mind, such as Scratch [12], Alice [13], Kodu [14], LEGO MINDSTORMS EV3 [15], and each language has its own unique structures, abstractions, and programming environment. One advantage of educational languages, from the user's perspective, is that syntax errors are often prevented, allowing the user to concentrate on program design and logic. For example, the blocks in Scratch and EV3 fit together like building blocks and the selection menu for choosing programming tiles in Kodu dynamically updates based on context. In addition to support to make programming easy, these languages also offer abstractions in the form of objects and subroutines, allowing programmers to create complex code [16]. As such, these languages, like professional languages, are susceptible to poor design and code smells. In this work, we focus on EV3 and Kodu, two popular educational languages with public repositories for sharing programs and with which the authors have prior experience. Next, we briefly introduce each language.

1) LEGO MINDSTORMS EV3: EV3 is the third iteration of the LEGO MINDSTORMS robotics line. It consists of several sensors, four motors and an ARM 9 "intelligent brick". The robotics kit comes with a control software package, which allows for visual programming of the brick, based on LabView[5]. The software supports several basic constructs common to programming, including loops and conditionals, as well as more advanced features like parallel execution. Figure 1 shows the user interface of the EV3 software with a program that makes a robot follow a black line by steering left if the sensor value is below the threshold of 7 and right if the value is above, and in parallel writes the sensor value to the screen. This program demonstrates some of the basic EV3 programming concepts including parallel execution, loops and switches. Blocks have to be connected to the 'Play button' on the left with wires to be executed. The block on the lower right is not connected and thus colored gray.

[4]We use the past tense as the Yahoo! Pipes environment has been discontinued as of September 30, 2015.

[5]http://www.ni.com/labview/

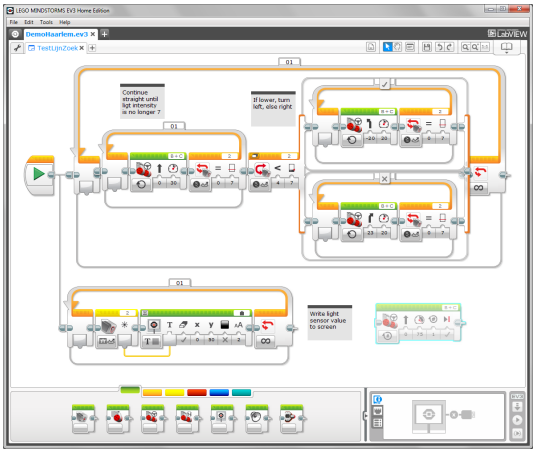


Fig. 1. The interface of LEGO MINDSTORMS EV3 showing a line following program.

Fig. 2. The interface of Kodu showing programming behavior for a bot that moves towards a red apple and then eats it.

In addition to the programming concepts described above, users have the possibility to define 'MY BLOCKS' which are basically subroutines. MY BLOCKS may have up to 9 different input parameters and one output parameter. MY BLOCKS cannot be programmed from scratch; they can only be created by selecting blocks in an existing program and clicking 'my block builder'. This puts the selected blocks in a new MY BLOCK and replaces the blocks with a call to the newly created MY BLOCK, an abstraction action very functionally similar to 'extract method' present in most modern IDEs.

2) Kodu: Microsoft Research's Kodu is a visual programming language [14] and environment that allows users to create, play, and share their own video games. It is available for download on the Xbox and PC and is heavily inspired by robotics, as illustrated by the context of programming in it and its language.

Users create a world (e.g., land, water, clouds), add characters and objects, and then can program each character or object (e.g., a kodu robot character, a turtle character, an apple object, a tree object; we use character and object interchangeably) individually. Variables are introduced as scores or properties of characters. Scores are integers and global variables, identified by color or letter, and readable and editable by all characters and objects. Character properties, such as color, glow color, expression and health, are like local variables. A character or object can read and write any of these properties for itself, and it can also read and write selected properties of other objects.

The programming defines how a character or object can interact with the world, treating each character or object as an autonomous agent. Each object has 12 pages that can be programmed, analogous to methods in an OO language, where the current page defines the current behavior of the object. The object's behavior can change by switching between pages to modify state and control flow. Each page contains a set of rules, and each rule is in the form of a condition and an action, which form a `when -- do` clause. The `when` is defined by a sensor (e.g., see, hear, gamepad input) and filters (e.g., apple, gamepad A button). The `do` is defined by an actuator (e.g., movement, shoot) and modifiers (e.g., missile, toward). All the rules on a page are evaluated in a single frame, from top to bottom. Kodu's unique language can be used to express many fundamental concepts in computer science, such as variables, boolean logic and conditional control flow [16].

Figure II-B2 shows a page and two rules in Kodu. For the first rule, the condition is, `when see red apple`, and the action is, `move toward it`. The action defines the behavior of this particular character when it sees a red apple, that is, it moves toward it. Since the condition identifies an object (i.e., apple), it becomes the default selector, even though it is not explicitly specified. The second rule, `when bump red apple, do eat`, has a similar condition with a different sensor, `bump`. The action of the second rule, `eat`, indicates that the character should eat any red apple it bumps. The programming in Figure II-B2 applies to the first page in this character's programming, as indicated by the number one at the top of the screen. The first page is the start page.

To form more complex boolean logic, rules can also be indented to create complex `when` clauses, where both conditions need to be true for the action to occur. Alternatively, indenting a rule and removing the `when` means that multiple `do` clauses occur for the same trigger condition.

III. Definitions of Smells

In Section II-A, we briefly summarized OO-inspired end-user smells from previous work. In both the Excel and Yahoo! Pipes work, it was found that OO-inspired smells were applied to end-user languages, and that some smells were found in both languages. This leads to the question how broadly applicable code smells are to end-user languages.

We begin by mapping OO smells to two education languages. Given that this is a first look into code smells in education languages, we limit our exploration to the 11 code smells found in other end-user programming languages, as shown in Table I. As is common in the other approaches, we define a loose mapping of OO concepts to the end-user languages in question. The thresholds we used for the study to determine when the presence of a pattern is considered smelly, (e.g., how big must a method be to have a *long method* smell?), are defined in Section IV.

A. Smells in LEGO MINDSTORMS EV3

In EV3 programming, MY BLOCKS abstract one or more blocks, can be called multiple times, and use input and output. As such they resemble methods in source code, modules in Yahoo! Pipes and worksheets in spreadsheets. Based on this translation, we investigate whether and how each of the smells in our catalog apply to EV3 programs.

Dead Code: It is possible for programming blocks to be disconnected, but the interface clearly indicates this as an issue. However, unused MY BLOCKS can be present in the project without a warning being issued. This is smelly as it makes the program unnecessarily large.

Deprecated Interface: This is a smell that does not apply, as, to date, there is only one version of the EV3 software and no blocks have been deprecated.

Duplicate Code: When the same or very similar combinations of blocks occur, the program suffers from the duplicate code smell.

Feature Envy: While all defined variables within EV3 programs are global, they can be written in a certain MY BLOCK but read in a different one. If, in a given MY BLOCK many variables are read that have been written somewhere else, this is be an occurrence of the feature envy smell.

Inappropriate Intimacy: Variables can be read in one MY BLOCK but written somewhere else. If there are two MY BLOCKS sharing multiple variables this way, it might be better to combine them and the program is smelly.

Lazy Class: If a MY BLOCK is very small, for example, consisting of just one block, it does not add a lot of value but impacte understandability as a user has to navigate to the MY BLOCK to see what its functionality is.

Long Method: If a MY BLOCK grows very large, it will no longer be easy to understand, counteracting the added value of the abstraction.

Many Parameters: MY BLOCKS can have 9 different parameters, which could be considered too much for easy understandability, especially since parameters need to be connected with wires, potentially leading to visual clutter.

Message Chain: Since MY BLOCKS can have both input and output parameters, they can form a message chain, in which values are continuously passed until they are used, while they could have been passed directly, outside of the MY BLOCKS.

No-op: It is possible to combine blocks in such a fashion that they do not actually contribute to the functionality of the program. For example, if a user stops the same motor twice, the second stop will be a no-op.

Unused Field: MY BLOCKS can define parameters, but the user is not forced to use them, hence creating a code smell.

B. Smells in Kodu

In Kodu, we consider pages of programming as analogous to methods or classes and scores and character properties as analogous to variables. These analogies are used in the following smell definitions:

Dead Code: If there exists a page with programming such that there is no explicit path of control flow from Page 1 to it, it is unreachable and therefore dead.

Deprecated Interface: Some language features may exist in early versions of Kodu but not be available in future versions. As none of these features were ever deployed, this smell is not possible.

Duplicate Code: Two pages for the same character with exactly the same set of rules, or two identical rules on a page constitute duplicate code. Alternatively, two rules on the same page with the same `when` clauses (i.e., sensor and filter) but different actions could be consolidated using the indent feature and thus are smelly.

Feature Envy: All global variables, such as game scores, can be read and written by any character. If a certain character reads variables that have been written by another character, this could be an instance of the feature envy smell.

Inappropriate Intimacy: A character has four local properties that describe its state: color, glow color, expressions (angry, crazy, happy), and health. If one character frequently checks the properties of another character, this could constitute inappropriate intimacy.

Lazy Class: If a character has no programming, it could be an instance of the lazy smell.

Long Method: A page with many, many rules may be difficult to understand. Some programming could potentially move to other characters.

Many Parameters: A game can have 37 different global scores. Games that use many of these could be unnecessarily complex and thus smelly.

Message Chain: It is possible for a character to create a chain of switches between pages without any logic on the page other than the jump. This would create a long and unnecessary message chain.

No-op: Jumping to a page with no logic is the logical equivalent of a null pointer. While no error would be raised, the character would no longer have any behavior and would be stuck. Alternatively, rules with `when` clauses but no `do` clauses do not perform any actions.

Unused Field: A global variable that is written to but not read is an instance of the the unused field smell.

IV. Study

The aim of this paper is to explore the following research question: **To what extent do code smells occur in programming languages aimed at education?** To answer this research question, we analyze and count code smells in programs created in two new domains: a data-flow educational language for programming robots, LEGO MINDSTORMS EV3 software, and an event-driven educational language for building and playing video games, Kodu. In this section, we describe the artifacts and analysis used for the study.

A. Artifacts

For each language, we sought to explore programs created by two communities, children (the intended audience of the languages) and the community at large, which often includes

children and adults. No demographic information was collected for any of the program authors in either language.

1) LEGO MINDSTORMS EV3: We gathered 17 programs from two data sources. The first source is a weekly robotics club for children aged 8 to 13. The first eight programs collected were designed to participate in two robot competitions: A sumo wresting game in which robot have to push each other out of a ring[6], and a search and rescue game in which robots have to detect a soda can[7]. To obtain a more diverse set of EV3 programs, we also solicited members of the EV3 programming group on Facebook to share their programs with us[8] and refer to these as programs from the *community*. Nine programs come from this second source.

2) Kodu: We gathered 27 programs from two data sources. For the first data source, we ran a workshop at the Microsoft FUSE lab that introduced children to Kodu Game Lab in a series of three 3-hour sessions. To recruit participants, we advertised the Kodu workshop using a mailing list of parents interested in Kodu. Children between the ages of 9 and 12 volunteered to participate, with parental consent. We collected 17 programs created during the workshop for analysis. For the second data source, we randomly sampled 10 programs from the public Xbox Live community.

B. Analysis

At a high level, EV3 is a robotics language and Kodu is inspired by robotics. As such, they share some high-level concepts, such as performing actions based on sensor values. This allows us to similarly tune the smell detection thresholds.

1) LEGO MINDSTORMS EV3: To count the number of instances of the *long method*, a smell was counted when the MY BLOCK had 10 or more blocks, as this is typically the size that all blocks would not fit on the screen anymore. For the *lazy class* we defined smelly as three blocks or fewer. For the *many parameters* smell, the use of four or more input values as typically in the programs one or two were used.

2) Kodu: To count the number of instances of the *long method*, a smell was counted when the page had 10 or more rules. For the *many parameters* smell, the use of four or more game scores. These thresholds were chosen because 10 rules cannot be viewed on a screen screen without scrolling and only a couple games used more than three game scores.

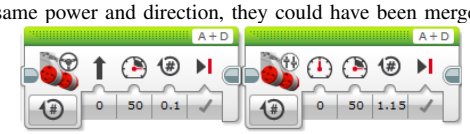
V. Results

In this section we explore the smell detection analysis in EV3 and Kodu, but summarizing the results, we found that most programs in both language suffer from at least one OO-inspired smell.

Table II summarizes the frequency of occurrence for each smell in the EV3 and Kodu programs. For each language and each code smell, we show the percentage of programs affected by each smell among programs created by children (i.e., the *Kids* column), those created by the community (i.e., the *Com* column), and overall (i.e., the *All* column). For example, 63% of the EV3 programs created by children have the *dead code* smell whereas only 11% of the EV3 programs created by the community have this smell. Among all EV3 programs we collected, 35% have the *dead code* smell.

Overall, 88% of the EV3 and 93% of the Kodu programs contained at least one smell. Coupled with the 81% of Yahoo! Pipes programs [9] and 42% of Excel programs [8] that were previously found to be smelly, this provides further evidence of the prevalence of smells in end-user programs.

Fig. 3. The duplication smell in two subsequent motor blocks. Since both have the same power and direction, they could have been merged.



A. LEGO MINDSTORMS EV3

On average, the programs from the children had 2.5 smells each and the programs from the community had 1.4 smells each, though the people submitting through the Facebook community most likely sent in nicely polished programs. Furthermore, the community programs were generally smaller (a median of 17 versus 23 total blocks) exhibiting more reuse as they more often used call to MY BLOCKS, though similar percentages of programs from the children and the community exhibited the duplicate code smell. There are only three smells that do not occur in any of the EV3 programs: *deprecated interface* (not applicable), *inappropriate intimacy* and *message chain*.

Next, we discuss the two most common smells within the EV3 programs, each appearing in at least one-third of the programs. The remaining smells appear in approximately one-fourth or fewer of the studied programs.

a) Duplicate Code: The most common smell we found in the programs is the *duplicate code* smell, which 65% of the programs suffer from. Duplication comes in various forms. Some of the programs use two motor blocks in a row, which could have been merged. For example, consider the two blocks from program L17, shown in Figure 3. This might have been challenging for the users to detect, as they use two slight variations of the motor block, but the two behave exactly the same in this case. Hence, one block moving forward 1.25 would have sufficed. Other programs exhibit duplication at a high level, such as the two MY BLOCKS in program L16, depicted in Figure 4. Here, the two MY BLOCKS perform the exact same operation, but on a different motor. By connecting the name of the motor to a parameter, like angle and speed already are, the same functionality could have been implemented with one MY BLOCK.

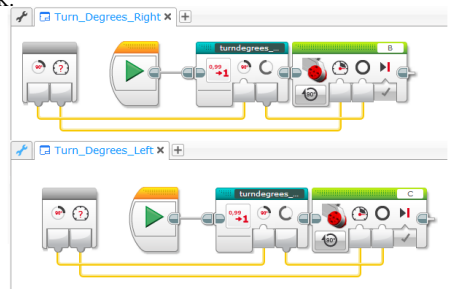
b) Dead Code: The second most common smell is *dead code*. In 35% of the programs we found MY BLOCKS that no longer connected to one of programs, though a disproportionate number of smelly programs were created by children (63%)

[6]http://www.sugobot.com/

[7]http://rcj.robocup.org/rescue.html

[8]https://www.facebook.com/groups/legomindstorms/permalink/527560164058881/

Fig. 4. The duplication smell in two different but very similar MY BLOCKS. By parameterizing the motor name, this could have been implemented with 1 MY BLOCK.



versus the community (11%). This smell can pose a problem, first for understandability, but also a more practical problem as the EV3 environment compiles and transfers all programs and MY BLOCKS to the physical brick, so dead code takes up unnecessary space on the brick. The *dead code* smells is more common in the robotics class programs than in the community programs. This is to be expected, as users felt confident enough to share their programs, so probably they either programs before sharing them, or selected 'clean' programs to share.

Looking at the EV3 programming interface, it is not that surprising that users forget about disconnected MY BLOCKS, as the EV3 interface provided no information on which MY BLOCK is used and where. Even worse, users can delete MY BLOCKS that are still being called, without a warning being issued about this. After removal of a MY BLOCK in use, the program no longer compiles.

1) Summary: Smells indeed occur in the EV3 programs. Overall, smells are found in 88% of the programs, with *duplicate code* and *dead code* appearing most frequently. Additionally, nearly one-fourth of the studied programs are affected by the *lazy class* (small abstractions) and *long method* smells, which may impact understandability. Hence we conclude that smells from OO are applicable to EV3.

B. Kodu

On average, the programs from children had 2.3 smells each and the programs from the community had 3.5 smells each, though we note that the programs from the community tend to be larger (e.g., an average of 101 rules vs. 42 rules). Next, we discuss the three most common smells found in the Kodu programs, appearing in one-third or more of the sample. The remaining smells appear in approximately one-fourth or fewer programs.

1) Lazy class: The *lazy class* smell was the most common, appearing in 81% of the Kodu programs. This smell is intended to capture when a character is placed in the world but has no programming, and thus no behavior. In some cases, this may not be a smell at all, such as when characters are used as decorations in the world (e.g., trees and rocks). In others, it could be similar to an object that is created but that has no behavior.

2) Duplicate Code: This smell was present in 63% of the programs. All instances were that of duplicate `when` clauses within a page, and there were no instances of identical rules on a page or identical pages within the same character. The prevalence of this smell shows a missed opportunity for consolidating the code. A higher percentage of this smell appeared in the programs from the community (80%) compared to the programs from children (53%). This may be due to differences in the sizes of the programs, since the community programs were over twice as big as the children programs, on average (101 rules vs. 42 rules).

TABLE II
SUMMARY OF SMELLS ACROSS EV3 AND KODU PROGRAMS

	EV3			Kodu		
	Kids	Com	All	Kids	Com	All
Dead Code	63%	11%	35%	6%	20%	11%
Deprecated	0%	0%	0%	0%	0%	0%
Duplicate Code	63%	67%	65%	53%	80%	63%
Feature Envy	38%	0%	18%	12%	40%	22%
Inappropriate	0%	0%	0%	6%	0%	4%
Lazy Class	50%	0%	24%	88%	70%	81%
Long method	13%	33%	24%	18%	40%	26%
Many Params	0%	11%	6%	0%	10%	4%
Message Chain	0%	0%	0%	0%	10%	4%
No-op	13%	11%	12%	35%	50%	41%
Unused Field	13%	11%	12%	24%	30%	26%
Any smell	88%	89%	88%	94%	90%	93%

3) No-op: The third most-common smell is the no-op, present in 41% of the programs. All instances of this smell were rules without `do` clauses, rather than jumping to pages without logic. This is probably the product of the rapid cycling between testing and developing observed in Kodu development [16]; since the clauses have no actionable logic, keeping them in the code does not impact the semantics of the program, though it may impact performance. We note that none of the rules following these no-op clauses were indented to create a conditional conjunction.

4) Summary: Smells indeed occur in the Kodu programs. Overall, smells are found in 93% of the programs, with *lazy class*, *duplicate code*, and *no-op* (small abstractions) appearing most often. Additionally, approximately 26% of programs are impacted by the *long method* and *unused field* smells. Hence we conclude that smells from OO are applicable to Kodu.

C. Summary

As with code written by other end-user programmers [9], *duplicate code* is prevalent in EV3 and Kodu programs, affecting over 60% of the samples in both languages. The other two smells that were studied in Excel and Yahoo! Pipes, *lazy class* and *long method*, are present in at least 24% of the studied EV3 and Kodu programs, illustrating commonality across all end-user programming domains.

Dead code is more frequently found in EV3 and *lazy class* smells are more common in Kodu. The *no-op* smell is present

in 41% of the Kodu programs, which is similar in spirit to the 41% of EV3 programs with *dead code*. *Feature envy* was also prevalent across both languages, appearing in 18% and 22% of the EV3 and Kodu programs, respectively, though in EV3, all instances of the *feature envy* smell are in the childrens' programs. We observe that the occurring smells seem to be anti-patterns that occur in programming in general, independent of the programming language.

VI. Beyond OO Smells

In the previous sections we have looked at OO smells in end-user programming languages. However, end-user programming environments offer many opportunities to define smells based on user behavior or unique elements of the domain. Hence in this section we explore opportunities for new smells in end-user domains that extend beyond the OO-inspired smells in Section II.

A. End-User Programming Smells

Considering all the end-user programming languages discussed in this paper, Yahoo! Pipes, Excel, EV3, and Kodu, the following smells appear in at least two languages but are not related to the OO smells.

1) Layout smells: The EV3 and Yahoo! Pipes languages are visual and involve connecting boxes with wires, making it easy to create an unwieldily structure. EV3 is especially problematic as it lacks an auto-layouting function.

A special case where layout issues arise is when a user creates a MY BLOCK. Remember that MY BLOCKS can only be created with an 'extract method' operation. If this operation is called, all selected are moved to the newly created MY BLOCK, and are replaced by one block that is a call to this new MY BLOCK. Other than that, the layout remains unchanged. So if a big MY BLOCK is created, the program will now have a big gap, making the program harder to read. This, and other layout smells too are unique to visual languages and can impede understandability.

2) Programming Organization smell: Communities of programmers often have standards for how to structure program code. These *population-based* smells were first introduced for Yahoo! Pipes [9] and can be uncovered by exploring programs written by the community. In Kodu, often the rules that map a character's behavior to user input from a gamepad, keyboard or mouse are at the top of a page and the remaining logic is at the bottom. Deviating from this structure can lead to programs that are more difficult to understand when shared. In spreadsheets, there have been attempts to define programming standards[9], if such a standard is common within a company, a deviation could be considered smelly as well.

a) Conflicting Outcomes smell: In robotics inspired languages, actions are performed based on sensor values. This is true of EV3 and Kodu and provides the opportunity to create conflicting outcomes for the same sensor. In Kodu, this can happen when the same `when` clause is followed by conflicting `do` clauses. For example, consider the following two Kodu rules:

```
(1) When hear turtle : do score 1 red
(2) When hear turtle : do unscore 1 red
```

The `do` clauses are in conflict, both scoring and unscoring the red variable score by 1. If the `do` clause on rule 2 was instead, `do score 1 red`, then the rules would both execute outcome would be adding 2 to the red score. In EV3, the same situation can occur, as it supports `wait` blocks, which are very similar to a `when` clause. For example, a user can define a `wait` block for a sensor to detect black. Only when the `wait` block returns true is the block that follows executed. When two similar `wait` blocks are defined with conflicting following blocks, for example one where the motor moves 60 degrees clockwise and one where it moves 60 degrees counterclockwise, they would conflict.

Flagging conflicting rules during development would help alert users to this smell. However, the likely outcome would be a fix that does not preserve the semantics of the program (unless the conflict was intentional), so this smell may actually be closer to a bug since the fix would not be a true refactoring.

B. Domain-Specific Smells

Each end-user domain has unique characteristics that create opportunities for defining domain-specific smells. In Yahoo! Pipes, prior work has explored the presence of broken or deprecated data sources as a smell unique to the domain [9]. In Excel, prior work defined referring to an empty cell assmelly [17]. Here, we explore smells in EV3 and Kodu that are unique to their languages and environments.

1) EV3: The LEGO MINDSTORMS EV3 programming language differs significantly from the other end-user domains, as EV3 programmers work with both software and hardware. While programming, users need to configure motor blocks to the right port, for example the blocks in Figure 3 are controlling motors A and D. This means that, when a user has the wrong mental model of what motor is connected to what port, the program will not function as wanted. It appears users run into this issue too; some of the programs we studied contained comment blocks in which the mapping was described, as an effort to ensure proper mapping. Hence a smell unique to the EV3 domain is a wrong mapping between the robot in reality and the motor and sensor blocks, forming a *hardware/software interface smell*.

2) Kodu: The Kodu environment involves designing games, programming games, and playing games. This, paired with the unique event-driven language, creates unique opportunities for domain-specific Kodu smells.

a) Single Input Device smell: Games in Kodu can be programmed and played using an Xbox controller or a keyboard and mouse. Many games are programed assuming a particular device for input, such as an Xbox. However, when playing a game in a different environment, that device may not be available, and thus the game is not portable. This smell could be mitigated by adding analogous rules for other input

[9]www.fast-standard.org/

devices, such as mapping the gamepad trigger switches to the keyboard shift keys or mapping the right stick to movement.

b) Lost Character smell: The *lazy class* smell is related to a non-programmed character or object. Characters and objects need to be clicked on to be programs. A design smell can then manifest when a programmed object cannot be easily found in the world, and thus there is program behavior that is difficult to change. Resolving this smell could involve physically moving the characters or objects to be more accessible, adding a color or speech bubble above them as a marker, or by adding an environment feature that maintains a list of programmed and non-programmed characters that can be accessed.

VII. Related Work

Related to the current research are efforts on code smells within traditional languages; we start with the work of Fowler [4]. His book gives an overview of code smells and corresponding refactorings. Recent efforts focused on the automatic identification of code smells by means of metrics. Marinescu [18] for instance, uses metrics to identify *suspect* classes, those classes that might have design flaws.

Within end-user programming, this paper builds upon two lines of related work. First, the work of Stolee and Elbaum that studied smells [9] and refactorings [6]. They designed a number of smells by transforming known OO smells to the domain of Yahoo! Pipes. The second direction is the work by Hermans *et al.* that also took OO smells as an inspiration for a number of smells in spreadsheets [7], [8]. Subsequently they described corresponding refactorings [19] and a tool that applies refactorings [20]. This final work generalized previous work by Badame and Dig [11].

In addition to the above described work on Yahoo! Pipes and spreadsheets, there is previous smells detection work on other end-user environments, too. For example, other researchers have studied performance smells in LabView, a visual language for system-design [10], [21].

Research related to EV3 and Kodu predominantly explores language design [16], [22]–[24] and educational benefits [25]–[28]. There is some, albeit limited, work on detecting code smells in educational languages. Chatley and Timbul describe Kenya, a simple programming language for educational purposes, which they have integrated into the Eclipse environment [29]. They built features that allow the detection of 'bad style' in programs to be detected and reported as code is written, concentrating on encouraging students to program like they were taught in class. This work might indicate that smell detection can be useful for education, but the authors did not evaluate the usefulness of this approach.

VIII. Threats to Validity

The threats to validity of this work inherit the threats to validity of the original studies [3], [6]–[9], [30] on end-user smells in spreadsheets and Yahoo! Pipes.

We have defined OO smells in two languages, EV3 and Kodu, yet we have not evaluated whether these smells matter to the end users. Future work is needed to determine the impact of these smell on users of the languages.

In Kodu, the presence of the *lazy class* smell may be intentional and simply represent a design decision (i.e., a rock is not programmed because it is there purely for aesthetics), and in EV3 the this smell might occur because users make up for the lack of ability to name single blocks. Evaluating these smells with actual users will help tease out when code is smelly and when it is actually designed intentionally.

We have sampled programs from two sources for each EV3 and Kodu, yet these programs may not be representative of what children and community members create. A larger-scale analysis is necessary to generalize these findings to the populations of EV3 and Kodu programs.

IX. Conclusion

This paper describes and studies code smells for an end-user programming domain not previously studied in code smell or refactoring research: educational languages. Specifically, we focus on two languages, LEGO MINDSTORMS EV3 and Kodu. We present an evaluation of these code smells in the form of two case studies over 44 end-user programs created by children and the community. The results show that indeed many of smells previously studied in end-user languages also apply in the new domains, and *lazy class* (small abstractions), *duplicate code* and *dead code* occur frequently in these educational languages. The contributions of this paper are:

- Definitions of end-user programming smells in EV3 and Kodu (Section III)
- Two case studies investigating end-user smells in educational programming languages: EV3 and Kodu (Section IV and Section V)
- Identification of future opportunities for domain-specific, non-OO-inspired smell detection in end-user programming domains (Section VI)

In the end, we observe that the original concept of code smells is applicable to end-user languages, of a quite different character than the textual languages aimed at professional developers for which the smells were originally defined. This underlines the applicability of these smells, and also warrants further research into issues end-users encounter while programming. For example, exploring the impact of code smells on children who use the educational languages or designing user-friendly refactoring tools for visual languages are potential future directions. Furthermore, studying the smells in a fresh context provides new insight on how to use smells in software engineering and could even give rise to new types of smells for professional languages.

Acknowledgements

Special thanks to Stephen Coy for his help with Kodu, to all children at 'Instituut het Centrum' and the members of Facebook group 'legomindstorms' for sharing their programs. This work is supported in part by NSF SHF-EAGER-1446932 and the Harpole-Pentair endowment at Iowa State University.

REFERENCES

[1] C. Scaffidi, M. Shaw, and B. A. Myers, "Estimating the numbers of end users and end user programmers," in *Proc. of VL/HCC '05*, 2005, pp. 207–214.

[2] A. J. Ko, R. Abraham, L. Beckwith, A. Blackwell, M. Burnett, M. Erwig, C. Scaffidi, J. Lawrance, H. Lieberman, B. Myers, M. B. Rosson, G. Rothermel, M. Shaw, and S. Wiedenbeck, "The state of the art in end-user software engineering," *ACM Computing Surveys*, vol. 43, no. 3, pp. 21:1–21:44, Apr. 2011.

[3] F. Hermans, M. Pinzger, and A. van Deursen, "Supporting professional spreadsheet users by generating leveled dataflow diagrams," in *Proc. of ICSE '11*, 2011, pp. 451–460.

[4] M. Fowler, *Refactoring: improving the design of existing code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[5] T. Mens and T. Tourwé, "A survey of software refactoring," *IEEE Trans. Soft. Eng.*, vol. 30, no. 2, pp. 126–139, Feb. 2004.

[6] K. Stolee and S. Elbaum, "Refactoring pipe-like mashups for end-user programmers," in *Proc. of ICSE '11*, 2011, pp. 81–90.

[7] F. Hermans, M. Pinzger, and A. van Deursen, "Detecting and visualizing inter-worksheet smells in spreadsheets," in *Proc. of ICSE '12*, 2012, pp. 441–451.

[8] ——, "Detecting code smells in spreadsheet formulas," in *Proc. of ICSM '12*, 2012, pp. 409–418.

[9] K. T. Stolee and S. Elbaum, "Identification, impact, and refactoring of smells in pipe-like web mashups," *IEEE Trans. Soft. Eng.*, vol. 39, no. 12, pp. 1654–1679, 2013.

[10] C. Chambers and C. Scaffidi, "Smell-driven performance analysis for end-user programmers," in *Proc. of VLH/CC '13*, 2013, pp. 159–166.

[11] S. Badame and D. Dig, "Refactoring meets spreadsheet formulas," in *Proc. of ICSM '12*, 2012, pp. 399–409.

[12] "Scratch," http://scratch.mit.edu/, February 2011.

[13] S. Cooper, W. Dann, and R. Pausch, "Alice: a 3-d tool for introductory programming concepts," in *CCSC '00: northeastern conference on The journal of computing in small colleges*, 2000, pp. 107–116.

[14] "Kodu language and grammar specification," http://research.microsoft.com/en-us/projects/kodu/grammar.pdf, August 2010.

[15] "LEGO Mindstorms," http://mindstorms.lego.com/, April 2015.

[16] K. T. Stolee and T. Fristoe, "Expressing computer science concepts through kodu game lab," in *Proceedings of the 42nd ACM technical symposium on Computer science education*, 2011, pp. 99–104.

[17] J. Cunha, J. P. Fernandes, H. Ribeiro, and J. Saraiva, "Towards a catalog of spreadsheet smells," in *Proc. of ICCSA '12*, 2012, pp. 202–216.

[18] R. Marinescu, "Detecting design flaws via metrics in object-oriented systems," in *Proc. of TOOLS '01*. IEEE Computer Society, 2001, pp. 173–182.

[19] F. Hermans, M. Pinzger, and A. van Deursen, "Detecting and refactoring code smells in spreadsheet formulas," *Empirical Software Engineering*, pp. 1–27, 2014.

[20] F. Hermans and D. Dig, "Bumblebee: a refactoring environment for spreadsheet formulas," in *Proc. of FSE '14*, 2014, pp. 747–750.

[21] C. Chambers and C. Scaffidi, "Impact and utility of smell-driven performance tuning for end-user programmers," *Journal of Visual Languages & Computing*, vol. 28, pp. 176–194, 2015, to appear.

[22] T. Fristoe, J. Denner, M. MacLaurin, M. Mateas, and N. Wardrip-Fruin, "Say it with systems: Expanding kodu's expressive power through gender-inclusive mechanics," in *Proc. of FDG '11*, 2011, pp. 227–234.

[23] M. MacLaurin, "Kodu: End-user programming and design for games," in *Proc. of FDG '09*, 2009, pp. 2:xviii–2:xix.

[24] M. B. MacLaurin, "The design of kodu: A tiny visual programming language for children on the xbox 360," *SIGPLAN Not.*, vol. 46, no. 1, pp. 241–246, Jan. 2011.

[25] A. Fowler and B. Cusack, "Kodu game lab: Improving the motivation for learning programming concepts," in *Proc. of FDG '11*, 2011, pp. 238–240.

[26] D. S. Touretzky, D. Marghitu, S. Ludi, D. Bernstein, and L. Ni, "Accelerating k-12 computational thinking using scaffolding, staging, and abstraction," in *Proc. of SIGCSE '13*, 2013, pp. 609–614.

[27] D. J. Barnes, "Teaching introductory java through lego mindstorms models," in *Proc. of SIGCSE' 02*, 2002, pp. 147–151.

[28] C. S. Hood and D. J. Hood, "Teaching programming and language concepts using legos," in *Proc. of ITiCSE '05*, 2005, pp. 19–23.

[29] R. Chatley and T. Timbul, "Kenyaeclipse: Learning to program in eclipse," in *Proc. of ESEC/FSE '05*, 2005, pp. 245–248.

[30] K. T. Stolee, J. Saylor, and T. Lund, "Exploring the benefits of using redundant responses in crowdsourced evaluations," in *Proc. of IW CSI-SE '15*, 2015, to appear.