A Catalog of OO-inspired End-User Smells, and Application to LEGO MINDSTORMS EV3 and Kodu

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

Kathryn T. Stolee
Department of Computer Science
Iowa State University
Ames, IA, U.S.A.
kstolee@iastate.edu

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

ABSTRACT

In the workforce today, millions of people program without degrees or professional training in software development. These end-user programmers perform a variety of tasks, from combining web information to building models to 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 recent research in two distinct end-user domains and languages: spreadsheets in Microsoft Excel and web mashups in Yahoo! Pipes. Based on existing OO-smells and their applications to these two end-user domains, we distill a catalog of generic end-user smells.

We demonstrate the broad applicability of the catalog by applying it to 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 indeed OO inspired smells occur in both end-user languages, most commonly we find that small abstractions, duplication and dead code are common. We conclude the paper by proposing new end-users smell, moving beyond the OO paradigm.

1. 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. While 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 code written by professional developers, end-user artifacts may have a long life-span, the average lifespan of a corporate spreadsheet being 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*.

The taxonomy of smells outlined in Fowler's text pertained to object-oriented (OO) code, and professional programming languages were the focus for at least the first decade of refactoring and code smell research [4], but in the past few years, research has also been done into smells in end-user programming. Most notable are structural smells in Yahoo! Pipes web mashups [5] and Excel spreadsheets [6]. Experiments in end-user areas have shown that end-user programmers understand smells, and often prefer versions of their code that are non-smelly [7–9].

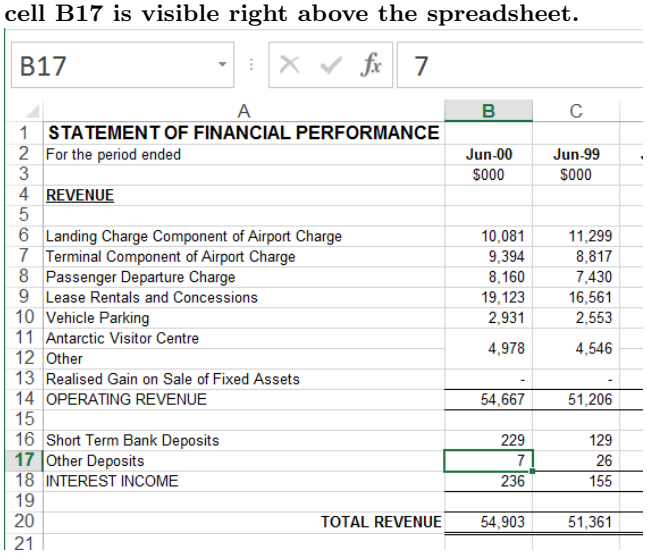
In this paper we combine the two end-user smell approaches and obverse that many of the OO smells are applicable in both domains, either they have been already studied, or they could be. This led us to the introduction of a generically applicable, OO-inspired end-user smells. To demonstrate the broad applicability of the catalog, we applied it to two new end-user domains aimed at education: LEGO MINDSTORMS EV3 and Kodu. The results of the application show that OO inspired smells from our catalog in fact occur in both end-user languages, underlining the power of the code smells concept: they also apply on visual languages aimed at education, which are quite different from the textual languages aimed at professional developers that the concept was initially designed for.

In EV3 and Kodu, the smells that we most commonly find are small abstractions (lazy class), duplication and dead code. In addition to the application of OO smells to the two new domains, we also move beyond the OO paradigm and hypothesize on domain specific smells for our four studied end-user languages: Excel, Yahoo! Pipes, EV3 and Kodu. The contributions of this work are as follows:

- Synthesis and catalog of object-oriented-inspired code smells in end-user programs
- Application of the catalog to two new end-user domains focused on education: LEGO MINDSTORMS EV3 and Kodu
- Identification of future opportunities for domain-specific, non OO-inspired smell detection in end-user programming domains

2. BACKGROUND

Figure 1: Microsoft Excel 2013 showing a spreadsheet. Column B and C shows the typical mix of data and calculation. The formula of the selected cell B17 is visible right above the spreadsheet.



B17 | 7

	A	B	C
1	**STATEMENT OF FINANCIAL PERFORMANCE**		
2	For the period ended	**Jun-00**	**Jun-99**
3		$000	$000
4	**REVENUE**		
5			
6	Landing Charge Component of Airport Charge	10,081	11,299
7	Terminal Component of Airport Charge	9,394	8,817
8	Passenger Departure Charge	8,160	7,430
9	Lease Rentals and Concessions	19,123	16,561
10	Vehicle Parking	2,931	2,553
11	Antarctic Visitor Centre	4,978	4,546
12	Other		
13	Realised Gain on Sale of Fixed Assets	-	-
14	OPERATING REVENUE	54,667	51,206
15			
16	Short Term Bank Deposits	229	129
17	Other Deposits	7	26
18	INTEREST INCOME	236	155
19			
20	**TOTAL REVENUE**	54,903	51,361
21			

In this section we briefly explore the two end-user languages targeted by prior smell research, before presenting the relevant end-user code smells in Section 3.

Excel: Spreadsheets are very commonly used in businesses, from inventory administration to educational applications and from scientific modeling to financial systems. Winston [10] estimates that 90% of all analysts in industry perform calculations in spreadsheets. Microsoft Excel is by far the most used, and therefore most studied, spreadsheet program, but other implementations exists and are similar.

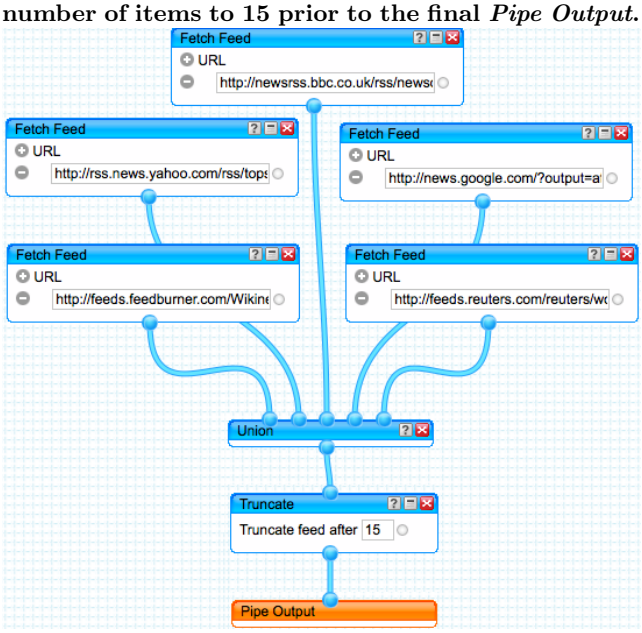
In modern spreadsheet programs, a *cell* can contain a single *formula* which performs a calculation, and a table of cells is bundled in a *worksheet*; an example is shown in Figure 1. A *workbook* consist of a collection of worksheets. Formulas can reference other cells in the same or in different workbooks and worksheets.

Yahoo! Pipes: Yahoo! Pipes is a popular web mashup language and environment with which RSS feed information can be collected and combined from various sources. Figure 2 shows an example program. The boxes represent modules connected by wires. Abstraction is possible with *subpipe* modules, which allow a programmer to insert a different pipe as a subroutine, appearing like a standard module.

3. SMELLS IN END-USER PROGRAMS

Research into end-user language smells has had two approaches, which are not mutually exclusive. The first approach is to take existing smells for OO programming languages, usually those defined by Fowler [11], and transform them to be applicable to the end-user environment [5–9]. 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 [9], by looking at user reports like forum or newsgroup posts [9,12], or by analyzing publicly available repositories [5,7,8].

Figure 2: Example of a program in Yahoo! Pipes. It has five RSS feed data sources, each in a *Fetch Feed* module, feeding to a *Union* module that concatenates the feeds, a *Truncate* module that limits the number of items to 15 prior to the final *Pipe Output*.



This section provides an overview of different smells that researchers have found to be applicable to end-user artifacts using both the above described approaches and proposes future directions for smell detection in these domains.

We summarize the OO smells present in two end-user languages, Excel spreadsheets and Yahoo! Pipes mashups, in Table 1. Overall, we often observe similarities in the code smells studied. For example, the *Duplicate Code*, *Lazy Class*, and *Long Method* smells have been studied in both languages. We note that the absence of an entry in Table 1 does not mean the absence of the smell in that language, it simply means such smells have not been studied. We speculate on a few of these opportunities in Section 3.3.

3.1 Excel

Hermans et al. [6,7] analogize a workbook to a program, a worksheet to a class inside that program and a cell to a method and use this to transform nine of Fowlers smells. An example of this is the *Long Method* smell, which translates to the *Multiple Operations* smell because formulas with a large number of operations suffer from similar problems as long methods. In an industrial case study with 10 spreadsheet users, the smells were found understandable by users, and with the smells errors in real-life spreadsheets were found.

3.2 Yahoo! Pipes

Stolee and Elbaum [5,8] treat a Yahoo! Pipes mashup as a class and each module as a method. Fields in a module are treated as parameters. Using this analogy, several OO smells were mapped to this language. The most common smell, appearing in nearly one-third of the 8,000 pipes studied, was *Duplicate Strings*, an instance of Fowler's *Duplicate Code* smell. *Duplicate Modules*, impacted nearly one-quarter of

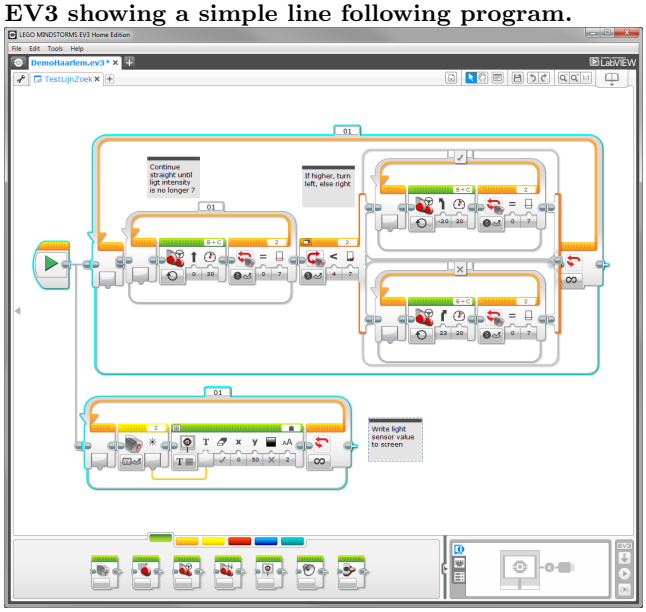
Table 1: Catalog of Code Smells in End-User Programs

OO Smell	Excel	Yahoo! Pipes
Dead Code		Unnecessary Module [8]
Duplicate Code	Duplicated Formulas [7]	Duplicate Modules, Duplicate String or Isomorphic Paths [8]
Feature Envy	Feature Envy [6]	
Inappropriate Intimacy	Inappropriate Intimacy [6]	
Lazy class or Middle Man	Middle Man [6]	Unnecessary Abstraction [8]
Long Method	Multiple Operations [7]	Noisy Module : Duplicate Field [8]
Many Parameters	Multiple References [7]	
Message Chain	Long Calculation Chain [7]	
No-op		Unnecessary Module [8]
Unused Field		Noisy Module : Empty Field [8]
Use of Deprecated Interfaces		Deprecated Module or Invalid Source [8]

the pipes studied. Overall, 81% of the programs studied from the Yahoo! Pipes community had at least one smell.

Yahoo! can also *deprecate modules*, which can create similar problems to using deprecated or old versions of interfaces.

Figure 3: The interface of LEGO MINDSTORMS EV3 showing a simple line following program.



3.3 Further Applications of OO smells

Although not all OO smells have been studied in both domains, indicated by empty places in Table 1, this does not mean they do not apply. For example, Feature Envy is not discussed in previous work on Yahoo! Pipes, but could apply when **TODO: Katie, how exactly?**. Similarly, redundant operations could occur in Excel, in fact, they do. A formula like SUM(A1+A2+A3), which we have seen in practice, exhibits the No-Op smell, as the SUM function does not add anything. While the occurrence of these smells falls outside of the scope of the current work, in this section we discuss the potential of generalizing some of the OO and domain-specific smell definitions to additional domains, thus addressing the empty places in Table 1.

3.3.1 *Excel*

Most of the smells studied in other end-user domains have been studied in Excel spreadsheets, but there is future potential in the areas of redundancy and deprecation. The *Dead code* smell

Furthermore, research in smells and spreadsheets has focused on Microsoft Excel. However, other spreadsheet software exists and operates on the same principles. Thus there is also an opportunity to confirm that the identified smells apply in other spreadsheet software.

3.3.2 *Yahoo! Pipes*

The *Feature Envy* smell could apply when introducing abstraction. For example, if a pipe has several instances of the same subpipe module, this could be excessive use of another class.

When a program uses too much abstraction relative to the size of the pipe, it could suffer from *Inappropriate Intimacy* by depending too much on the implementation of the other class. In fact, in an empirical evaluation, programmers often preferred pipes without subpipe modules because they were easier to understand [8].

A *Long Module* smell could apply when a module has a large number of fields. For example, the *Fetch Feed* module, as in Figure 2, can hold one or more URLs. When the number of URLs makes the method so big it does not fit on the screen, this would likely impact the understandability of the pipe.

Drawing inspiration from the domain-specific *Inconsistent Formula* smell in Excel, identifying program patterns that are close, but not exactly the same, could identify missed

opportunities for abstraction or errors in the mashup structure.

4. APPLICATION

In Section 3, we distill a catalog of end-user smells from previous work. In order to demonstrate the applicability of our catalog, we apply it to two new domains: LEGO MINDSTORMS EV3 software (EV3 for short) and Kodu.

4.1 LEGO MINDSTORMS EV3

LEGO MINDSTORMS EV3 is the third iteration of the LEGO MINDSTORMS robotics line. It consists of a number of 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. The software supports several basic constructs common to programming, including loops and conditionals, but also more advanced features like parallel execution. Figure 3 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. Blocks have to be connected by wires, like the two coming from the 'Play button' that acts as the starting point, to be executed. The program in Figure 3 demonstrates some of the basic EV3 programming concepts including parallel execution, loops and switches.

In addition to the programming concepts described above, users have the possibility to define 'MY BLOCKS' which basically are subroutines. MY BLOCKS may have up to 9 different input and one output parameter. MY BLOCKS cannot be programmed from scratch; they can only be created by selecting blocks in an existing program and creating a new block for them, together with a call to the newly created block, an action extremely similar to 'extract method' present in most modern IDEs.

4.1.1 Smells in LEGO MINDSTORMS EV3

In this subsection we describe how the smells in our OO-inspired catalog apply to EV3 programs. As common in the other approaches, we define a loose mapping of OO concepts to the end-user language in question to be able to translate the smells. In EV3 programming, there are MY BLOCKS that contain a number of blocks, can be used multiple times and can 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 which of the smells in our catalog could apply to EV3 programs too. **TODO: in what order should we present the smells? I think alphabetical makes more sense than no ordering?**

Dead Code It is possible for programming blocks to be disconnected, but the interface clearly indicates this by making them gray **TODO: added picture here?**. 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 Interfaces 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 depreciated.

Duplicate Code When the same, or very similar combinations of blocks occur, this would be 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 might be an occurrence of the feature envy smell.

Inappropriate Intimacy Variables might 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.

Lazy Class If a MY BLOCK is very small, for example, consisting of just one block, they do not add a lot of value, while making the program harder to understand, as a user has to navigate to the MY BLOCK to see what its function 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.

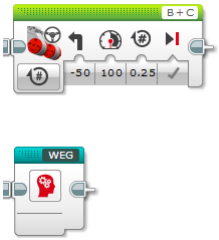
Many Parameters MY BLOCKS can have 9 different parameters, which could be considered too much for easy understandability, especially since parameters need to be connected with wires, potentially leading to visual clutter in the case of many parameters.

Message Chain Because MY BLOCKS can have both input and output parameters, it is possible that they created a message chain in which values are continuously passed until they are used, while they could have been passed 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, it a user stops the same motor twice, the second stop will be a no-op.

Unused Field As explained above, MY BLOCKS can define parameters. However, the user is not forced to use them, hence it is possible to define more parameters than used.

Figure 4: Two blocks representing the same functionality, one in a regular block and one using a my block.



4.1.2 Study context

To determine if EV3 programs indeed suffer form the above defined smells, we have gathered 17 programs from two data sources. The first source is robotics club ran by one of the authors of this paper. It is a club where kids aged 8 to 13 program robots every week. Their programs have been collected as dataset for this paper. More specifically, we focus on two different projects within this set, RoboCup and Sumo. These two types of programs related to two different LEGO MINDSTORMS competitions: Sumo is a simple robot game in which robots have to 'sumo wrestle' each other: the robot that gets pushed out of the circular competition area first loses[1]. The Robocup programs were made to participate in the RoboCup Junior Rescue challenge[2], where robots have to first navigate part of the field by following a line and subsequently look for a soda can and push it out of the field.

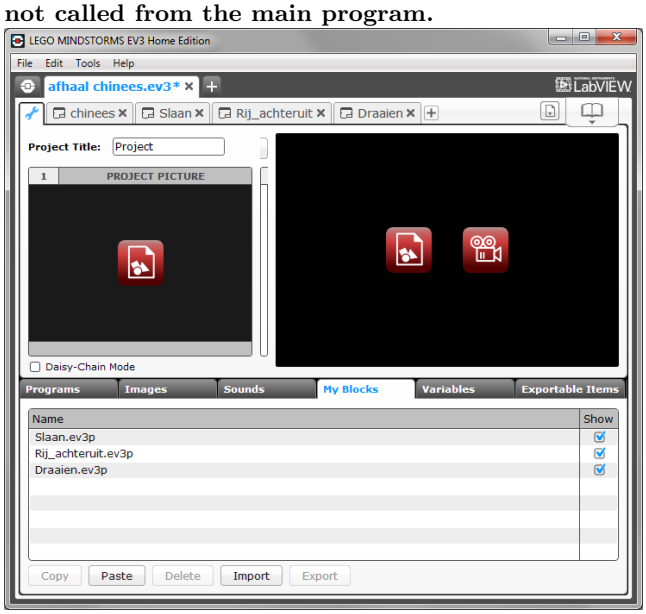
To obtain a more diverse set of EV3 programs, we have solicited members of the EV3 programming group on Facebook to share their programs with us[3].

4.1.3 Findings

When investigating the programs, we found that they indeed can suffer from OO-inspired smells. There are only two smells that do not occur in any of the programs: Inappropriate Intimacy and Message Chain. Duplicate Code occurs most, in over half of the programs.

Table 2 presents an overview of the smells found in the EV3 programs.

Figure 5: The project properties screen for the Sumo program 1, showing the three my blocks , but not indicating the my block 'draaien' is currently not called from the main program.



[1] http://www.sugobot.com/
[2] http://rcj.robocup.org/rescue.html
[3] https://www.facebook.com/groups/legomindstorms/permalink/527560164058881/

Duplicate Code.

The most common smell we found in the nine programs is the Duplicate Code smell, which 11 out of 17 programs suffer from. Duplication comes in various forms, some of the programs use two motor blocks in a row, that could have been merged, like the two blocks from program L17, shown in Figure 6. 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 special case, where the direction is straight forward. Other programs exhibit duplication at a high level, like two MY BLOCKS in program L16, depicted in Figure 7. 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 two, this functionality could have been implemented with one MY BLOCK.

Figure 6: The duplication smell in two subsequent motor blocks. Since both have the same power and direction, they could have been merged.

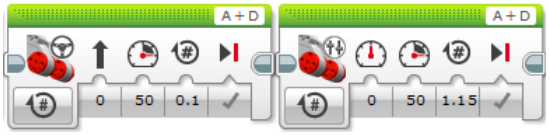
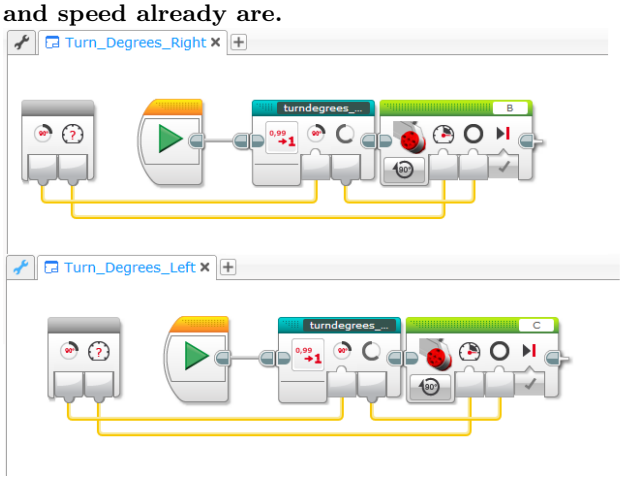


Figure 7: The duplication smell in two different my blocks. Both my blocks turn a motor, and which motor could have been put in a parameter, like angle and speed already are.



Dead Code.

The second most common smell is Dead Code. In seven out of 17 programs we found MY BLOCKS that no longer connected to one of programs. This can pose a problem, as the EV3 environment compiles and transfers all programs and MY BLOCKS to the brick, causing the memory to be full quite quickly. We see that Dead Code is more common in

the robotics class programs than in the programs we received via Facebook. This is probably due to the fact that these users felt confident enough to share their programs, because either the programs were reviewed before sharing them, or the experience users selected 'nice' programs to share.

Looking at the EV3 programming interface, it is not that surprising that users forget about disconnected MY BLOCKS. The interface does not help users in understanding what MY BLOCKS are used. If we look at the project properties screen, shown in Figure 5, one can see that there is no information about which MY BLOCK is used where. Even worse, user 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.

Lazy Class.

Lazy classes, which we counted as MY BLOCKS containing three or fewer blocks, are also relatively common, occurring in four of the programs. However, we only find this smell in the programs from the robotics club and not in the programs received via Facebook. Many of 'lazy MY BLOCKS' were relatively small, consisting of two or three blocks, or even one in some cases. You could say this is smelly, as understanding the MY BLOCK requires clicking it and that might not be worth it for small MY BLOCKS. However, the EV3 programming interface does not allow regular blocks to be named, but it does allow this for MY BLOCKS. So by making a MY BLOCK, users can express the intent of a coherent set of blocks, even if this set consist of just one block.

As an example, consider the two blocks shown in Figure 5, where the upper one is regular block controlling a motor, while the lower one is a call to a MY BLOCK with the same functionality. The first one just expresses what the robot must do, but the second one expresses the intent ('weg' meaning flee). By using the MY BLOCK, even with one block, the program gets easier to read.

Long method.

Long Method **TODO: add threshold here** too occurs in four programs, but, contrary to Lazy Class, is found mostly in the Facebook programs. This is to be expected, more experienced users can handle bigger MY BLOCKS, while the kids from the robotics club stick with making small blocks.

4.1.4 Summary

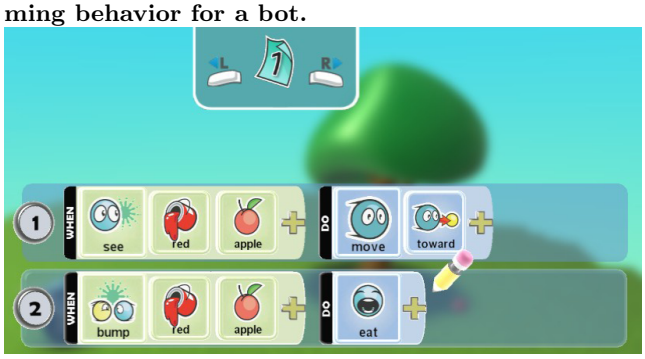
To summarize, we again observe that smells stemming from OO are applicable to end-user domain: duplicate code, dead code and small abstractions seem to be anti-patterns that occur in programming in general, independent of the programming language. It is remarkable to see that even in a visual language, quite different from the textual, high-level languages for which the smells were originally designed, the same smells occur.

4.2 Kodu

Microsoft Research's Kodu is a visual programming language [13] 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, and the context of programming in it and its language both show this. Users can program each character or object (e.g., a kodu, cycle, apple, tree; we use character and object interchangeably) individually, and the programming defines how to interact with the world. 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. Despite its unique language, Kodu can be used to express many basic concepts in computer science, such as variables, boolean logic and conditional control flow [14].

Figure 8 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 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. 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. The programming in Figure 8 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 default start page.

Figure 8: The interface of Kodu showing programming behavior for a bot.



4.2.1 Smells in Kodu

We describe how the smells in our OO-inspired catalog apply to Kodu programs. We define a loose mapping of the OO concepts to the end-user language. We consider pages of programming as analogous to methods or classes, similar to how modules are treated in Yahoo! Pipes, worksheets in spreadsheets and MY BLOCKS in LEGO MINDSTORMS EV3.

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.

Table 2: Size in terms of the different number of blocks used in the seventeen LEGO MINDSTORMS EV3 programs, and the smells they exhibit TODO: deprecated interfaces?

Name	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L11	L12	L13	L14	L15	L16	L17
Data	1	0	2	5	0	0	4	4	1	0	0	2	0	7	1	1	0
Actor	8	6	5	7	16	8	10	16	10	1	2	1	7	15	2	2	78
Sensor	1	0	0	2	1	0	8	10	4	1	0	1	0	4	2	0	0
Logic	3	5	4	4	4	3	7	11	2	2	1	4	11	2	0	0	25
MyBlock Call	2	0	2	2	3	4	3	5	0	1	0	1	0	3	1	4	27
Comment	2	0	0	1	0	0	1	0	0	6	6	0	15	0	0	7	13
Variables	0	0	1	3	1	1	0	0	0	0	0	0	0	19	0	0	0
MyBlocks	3	0	3	2	3	2	2	4	0	1	0	1	0	4	1	3	6
Total	20	11	17	26	28	18	35	50	17	12	9	10	33	54	7	17	149
Dead Code	✓		✓	✓			✓	✓						✓			✓
Deprecated																	
Duplicate Code				✓	✓	✓	✓	✓	✓	✓	✓		✓			✓	✓
Feature Envy	✓		✓	✓													
Inappropriate Intimacy																	
Lazy Class	✓		✓	✓		✓											
Long method					✓					✓				✓			✓
Many Parameters												✓					
Message Chain																	
No-op			✓											✓			
Unused Field				✓										✓			
Total Smells	3	0	4	5	2	2	2	2	1	2	1	1	1	4	0	1	3

Deprecated Interfaces Some language features and bots may exist in early versions of Kodu but not be available in future versions. use of these deprecated entities may cause unexpected behavior when running an old program in a new environment and be smelly.

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). 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 pages or other characters. We counted long methods as those with more than five rules.

Many Parameters A game can have 37 different global scores. Games that use many of these could be unnecessarily 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 in that state. 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.

4.2.2 Study Context

To determine if Kodu programs indeed suffer form the above defined smells, we have gathered 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. No demographic information was collected about the users.

The programs were analyzed by hand by one of the authors. Table 3 presents the 17 programs created by children (K1 ... K17), 10 programs created by the Xbox Live community (K18 ... K27), and the smells found in each. The *Characters* row defines how many characters and actors are in the program, *Pages* defines how many pages of programming are used by the characters, *Rules* counts all the rules on all the pages, and *HCI Actors* counts the number of actors

Table 3: Size in terms of the different number of characters and rules used in the 17 Kodu programs, and the smells they exhibit

Name	K1	K2	K3	K4	K5	K6	K7	K8	K9	K10	K11	K12	K13	K14	K15	K16	K17	K18	K19	K20	K21	K22	K23	K24	K25	K26	K27
Characters	6	17	39	21	62	60	85	26	26	60	42	45	57	36	5	15	28	28	10	2	16	18	76	75	63	4	76
Pages	11	7	26	19	18	47	82	19	10	60	41	31	10	9	6	7	7	34	16	2	39	18	9	50	90	8	78
Rules	21	17	60	28	31	96	93	29	20	120	58	63	22	35	14	18	23	114	54	3	113	27	30	122	144	32	372
HCI Actors	1	2	1	1	2	1	1	1	1	1	1	1	1	0	2	1	1	5	1	0	0	1	1	1	37	0	2
Dead Code													✓											✓	✓		
Deprecated																											
Duplicate Code			✓			✓	✓	✓	✓			✓	✓	✓		✓		✓	✓	✓	✓	✓		✓	✓		✓
Feature Envy			✓							✓									✓					✓	✓		✓
Inapprop. Int.																✓											
Lazy Class		✓	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓		✓	✓		✓	✓	✓		✓
Long method						✓						✓	✓					✓	✓					✓			✓
Many Params																											✓
Message Chain																								✓			
No-op		✓								✓		✓	✓	✓			✓		✓	✓				✓	✓		✓
Unused Field			✓						✓			✓			✓			✓							✓		✓
Total Smells	0	2	4	1	1	3	2	2	3	2	1	5	5	3	2	3	2	4	4	3	2	1	1	7	6	0	7

in the program that are controllable by the Xbox controller, mouse or keyboard. 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 determined by the authors as 10 rules cannot be viewed on a screen screen without scrolling and only a couple games used more than three game scores.

4.2.3 Findings

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 40% or more of the sample. The remaining smells appear in 25% or fewer programs.

4.2.4 Lazy class

The lazy class smell was the most common, appearing in 79% 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.

4.2.5 Duplicate Code

This smell was present in 61% 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 feature shows a missed opportunity for consolidating the code.

4.2.6 No-op

The next 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 [14]; since the clauses have no actionable logic, keeping them in the code does not impact the semantics of the program.

4.3 Summary

Table 4 summarizes the frequencies of occurrence of the various smells in the EV3 and Kodu programs. Overall, 88% of the EV3 and 93% of the Kodu programs contained at least one smell. Coupled with the 81% of Yahoo! Pipes programs [8] and **TODO: X% of Excel programs** that were previously found to be smelly, this provides further evidence of the prevalence of smells in end-user programs.

As with code written by other end-user programmers [8], duplication smells are prevalent in EV3 and Kodu programs, affecting over 60% of the samples in both languages. Dead code is more frequently found in EV3 and lazy classes are more common in Kodu. Long methods are found in approximately 25% of the programs in both languages, and **TODO: Y% of the Excel programs**, showing that this smell, too, is common across languages. Feature envy was also prevalent across both languages, appearing in 18% and 22% of the EV3 and Kodu programs, respectively.

5. BEYOND OO

The end-user programming environments offer many opportunities to define smells based on user behavior or unique elements of the domain. Here, we explore opportunities for new smells in end-user domains that extend beyond the OO-inspired smells in Section 3.

5.1 Excel

In spreadsheets, data is often processed by having each record in a row and having a column with identical formulas to perform some calculation. This leads to the definition of the *Inconsistent Formula* smell, which occurs if a single, or small number, of cells contain a different formula while its neighbors or other cells in the row or column contain an identical formula. Interestingly this smell is already detected by Microsoft Excel which warns the user about it.

Table 4: Summary of Smells Across EV3 and Kodu Programs

	EV3	Kodu
Dead Code	41%	11%
Deprecated Interfaces		0%
Duplicate Code	65%	61%
Feature Envy	18%	22%
Inappropriate Intimacy	0%	4%
Lazy Class	24%	79%
Long method	24%	25%
Many Parameters	6%	4%
Message Chain	0%	4%
No-op	12%	41%
Unused Field	12%	25%
Any smell	88%	93%

Another smell specific to spreadsheets is when a formula references an empty cell, which is often in error [15]. This is comparable to a null pointer in other languages and would be a runtime error, but because a spreadsheet contains both the input data and logic we can directly mark it as a smell.

5.2 Yahoo! Pipes

By exploring a large subset of the Yahoo! Pipes repository, Stolee and Elbaum identified a smell based on the presence of broken data sources [8]. A reference to a broken data sources is similar to opening a non-existing file, which would be a runtime exception or error in most professional languages. Such exceptions are not in the Yahoo! Pipes language, which is the reason to mark it as a smell. Similar exploration was used to identify common programming practices, marking deviations from those practices as smells. This is similar to identifying smells as anti-patterns and could be extended to any language.

5.3 EV3

TODO: Katie, please check if the following makes (enough) sense without knowing EV3, I am too deep in to be able to judge

5.3.1 Wrong motor or sensor mapping

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 6 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, as some of the programs we saw comment blocks where the mapping was described **TODO: if room add example**.

Hence a smell unique to the EV3 domain is a wrong mapping between the robot in reality and the motor and sensor blocks.

5.3.2 Layout smells

The EV3 language is a visual language without an auto-layouting function, this means that, when users are making changes to their programs, often the layout can get messy. This also happens when a user creates a MY BLOCK, blocks that have been moved to the MY BLOCK are replaced by a call to the newly created MY BLOCK, but 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 **TODO: if room show example**. This, and other layout smells too are unique to visual languages where the layout is not modified by the environment.

5.4 Kodu

6. RELATED WORK

Related to the current research are efforts on code smells within traditional languages, starting with the work of Fowler [11]. 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 [16] for instance, uses metrics to identify *suspect* classes, those classes that might have design flaws. Lanza and Marinescu [17] explain this methodology in more detail. Alves *et al.* [18] focus on a strategy to obtain thresholds for metrics from a benchmark. Olbrich *et al.* furthermore investigates the changes in smells over time, and discusses their impact [19].

Within end-user programming, this paper builds upon two lines of related work. Firstly, the work of Stolee and Elbaum that studied smells [8] and refactorings [5]. 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 from a number of smells [6, 7]. Subsequently they described corresponding refactorings [20] and a tool that applies refactorings [21]. This final work builds upon previous work by Badame and Dig that described the first spreadsheet refactoring tool RefBook [12].

In end-user programming environments, user input and logic are often more closely linked than they are in general purpose languages, and as such analyzing the input data as opposed to the logic can also be used as a means of *smell detection*. This is a direction that has been spplied to spreadsheets. Cunha et. al [15] for example look at anomalies in the data and define these as smells. Examples of this are *Standard Deviation* which occurs if one assumes a normal distribution for a column in numeric values and the column contains values which fall outside two standard deviations. In more recent work, Barowy et. Al [22] take a more formalized approach which they label "Data Debugging". Their solution uses statistical analysis to find values with an unusually high impact on the calculated results in a spreadsheet, as such values are likely either very important or erroneous.

In addition two above described work on Yahoo! Pipes and spreadsheets, there is previous smells work on other end-user environments too, like performance smells in LabView, a visual language for system-design [9, 23].

Finally, there is some, albeit limited, work on detecting code smells in educational languages. **TODO: Katie, do you know of more?** Chatley and Timbul describe Kenya, a simple programming language for educational purposes, which they have integrated into the Eclipse environ-

ment [24]. 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 unfortunately did not perform an evaluation of the usefulness of this approach.

7. DISCUSSION

Based on the research and results for smell detection in end-user programming domains, there are many directions for future work in the domains studied and other end-user domains.

7.1 Threats to Validity

The threats to validity of this work inherit the threats to validity of the original studies [3, 5–8, 12, 21, 25] on end-user refactoring in spreadsheets and Yahoo! Pipes.

The three domains studied in this paper all happen to be dataflow languages, and the smells and refactorings may not generalize to other end-user programming domains (e.g., Scratch is OO-based). **TODO: katie, would you say Kodu is dataflow based too?**

TODO: In Kodu, smells vs. design decisions

8. CONCLUDING REMARKS

This paper presents an overview of the work in smell detection for end-user programming languages. More specifically, it synthesizes work on Yahoo! Pipes and Excel into a catalog of generally applicable smells in end-user languages. To demonstrate the applicability of the catalog, we apply it to two new domains: LEGO MINDSTORMS EV3 and Kodu, two visual languages aimed at programming education. The results show that indeed many of the catalog's smells apply in the new domains, and small abstractions, duplicate code and dead code occur in these educational languages too, while being of a quite different character than the textual languages aimed at professional developers that the smells were originally defined for. This shows the applicability of these smells, and also warrants further research into issues end-users encounter while programming. The contributions of this paper are:

- A catalog of object-oriented-inspired code smells in end-user programs (Section 3)
- Application of the catalog to two new end-user domains focused on education (Section 4)
- Identification of future opportunities for smell detection in end-user programming domains, both within and beyond the OO paradigm (Section 5)

The current work gives rise to more research, for example **TODO: future work**

The applicability of smells, originally created to detect weaknesses in source code, to other domains shows how powerful the concept is. Furthermore, studying the smells in a fresh context provides new insight on how to use smells in software engineering and even suggests new types of smells.

Acknowledgements

Special thanks to Stephen Coy for his help with Kodu. This work is supported in part by NSF SHF-EAGER-1446932 and the Harpole-Pentair endowment at Iowa State University.

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

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

[6] 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.

[7] ——, "Detecting code smells in spreadsheet formulas," in *Proc. of ICSM '12*, 2012.

[8] 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.

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

[10] W. Winston, "Executive education opportunities," *OR/MS Today*, vol. 28, no. 4, 2001.

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

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

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

[14] 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*, ser. SIGCSE '11, 2011, pp. 99–104.

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

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

[17] M. Lanza, R. Marinescu, and S. Ducasse, *Object-Oriented Metrics in Practice.* Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005.

[18] T. L. Alves, C. Ypma, and J. Visser, "Deriving metric thresholds from benchmark data," in *Proc. of ICSM '10*. IEEE Computer Society, 2010, pp. 1–10.

[19] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka, "The evolution and impact of code smells: A case study of two open source systems," in *Proc. of ESEM '09*, 2009, pp. 390–400.

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

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

[22] D. W. Barowy, D. Gochev, and E. D. Berger, "Checkcell: data debugging for spreadsheets," in *Proc. of IC OOPSLA '14*. ACM, 2014, pp. 507–523.

[23] 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.

[24] R. Chatley and T. Timbul, "Kenyaeclipse: Learning to program in eclipse," in *Proc. of ESEC/FSE '05*, ser. ESEC/FSE-13. New York, NY, USA: ACM, 2005, pp. 245–248. [Online]. Available: http://doi.acm.org/10.1145/1081706.1081746

[25] 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.