Smell Detection and Refactoring in End-User Programming Languages

Felienne Hermans, David Hoepelman
Delft University of Technology
Mekelweg 4
2628 CD Delft, the Netherlands
f.f.j.hermans@tudelft.nl

Kathryn T. Stolee
Iowa State University
209 Atanasoff Hall
Ames, IA 50011-1041
kstolee@iastate.edu

***Abstract*—In the workforce today, millions of people program without degrees or professional training in software development. These end-user programmers write code that designs hardware circuits, collects web information, and impacts business decisions. Software engineering research, an in particular refactoring, has traditionally focused on professional, and often object-oriented, programming languages, Yet, other domains and languages also suffer from code smells in need of refactoring. In this work, we explore recent research in three end-user domains and languages, spreadsheets in Microsoft Excel, web mashups in Yahoo! Pipes, and system designs in National Instruments' LabVIEW. Through exploring the commonalities and differences among the domains, we 1) show how these end-user domains benefit from prior research in refactoring for object-oriented languages, 2) discuss unique smell detection and refactoring opportunities for these domains and how they can translate to professional languages, and 3) identify future opportunities for smell and refactoring research in these studied domains as well as other domains of end-user programming languages.**

***Keywords*-code smells; end-user programming; refactoring;**

I. Introduction

The number of end-user programmers is said to greatly exceed the number of professional programmers. "The proportion of American end user workers reporting they *do programming* has remained relatively constant, rising from around 10% in 1989 to only around 15% in 2001" [1] This 15% of 2001 totals to about 11 million end users, while the number of professional developers was estimated at 3 million for the same period [1].

End-user programmers perform a wide variety of tasks within their organizations, ranging from building or maintaining applications to simple data manipulation 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. In fact, a 2011 case study showed that spreadsheets have an average lifespan of 5 years [3]. During this long lifespan, end-user artifacts are modified, often by different people. This makes them, like source code artifacts, vulnerable to *smells*.

Smells in end-user programming have been a topic of research over the past few years. Most notable are structural smells in Yahoo! Pipes web mashups [4] and Excel spreadsheets [5] and performance smells in LabVIEW code [6]. Experiments in all these areas have shown that end-user programmers understand smells and often prefer versions of their code that are non-smelly. Alleviating those smells can be achieved with refactoring.

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. This is what we study in this paper. The contributions of this work are:

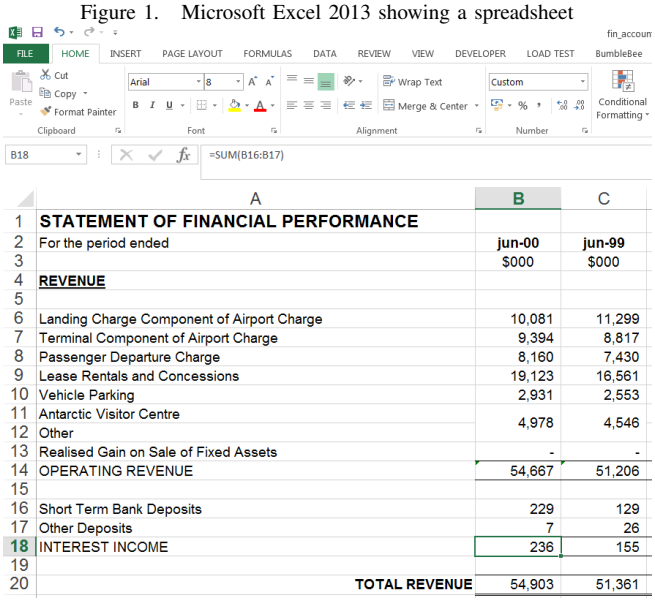
- Synthesis and catalog of object-oriented-inspired code smells and refactoring in end-user programs
- Discussion of how smells and refactoring in end-user domains may translate to professional languages
- Identification of future opportunities for smell detection and refactoring in end-user programming domains

II. Background

Refactoring was first introduced as a systematic way to restructure source code and facilitate software evolution and maintenance [7], [8]. In 1999, Martin Fowler introduced the concept of code smells for object-oriented (OO) languages [9]. According to Fowler, code smells indicate suspicious or weak parts that the developer might want to change in order to improve readability and minimize the chance of future errors. "A code smell is a surface indication that usually corresponds to a deeper problem in the system," says Fowler. For professional programmers, refactoring code is usually motivated by noticing a code smell, which signals the opportunity for improvement.

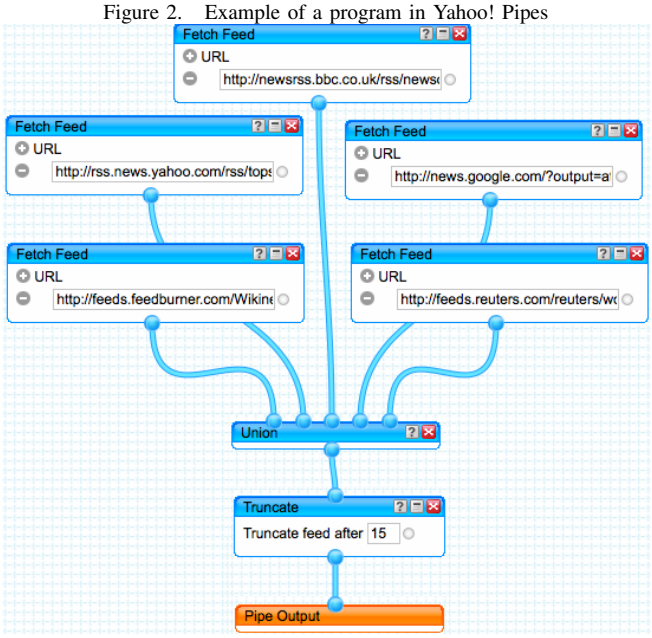
The taxonomy of smells outlined in Fowler's text pertained exclusively to object-oriented code, and professional programming languages were focus for at least the first decade of refactoring and code smell research [10]. Since 2011, however, refactoring and smell definitions have been adapted and extended to other programming language

Figure 1. Microsoft Excel 2013 showing a spreadsheet



	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

Figure 2. Example of a program in Yahoo! Pipes



paradigms, including web mashups [4], [11], Excel spreadsheets [3], [5], [12], and LabVIEW programs [6], all of which are end-user programming domains.

More importantly, researchers have found that code smells matter to end-user programmers. When presented with Yahoo! Pipes web mashups with and without smells, a significant majority of end-user programmers preferred the pipes without smells [11]. In a field study done with 10 spreadsheet users at an investment bank, users agreed smells could reveal actual errors in their spreadsheets**TODO: []**. When presenting LabVIEW users with information about code smells in their programs, all 13 participants were able to identify the problems and 12 were able to fix the problem [6]. Considering the large number of end-user programmers, the longevity of their artifacts, and the real impact of smells on understandability, errors, and performance, supporting end-user programmers in code smell detection and refactoring is valuable.

In this work, we explore past research in three end-user programming domains and languages: spreadsheets in Microsoft Excel, web mashups in Yahoo! Pipes, and visual system design in National Instruments' LabVIEW. Furthermore we identify future opportunities for smell and refactoring research in these and other end-user programing domains. Here, we briefly explore each language before presenting the relevant code smells in Section III and refactorings in Section IV.

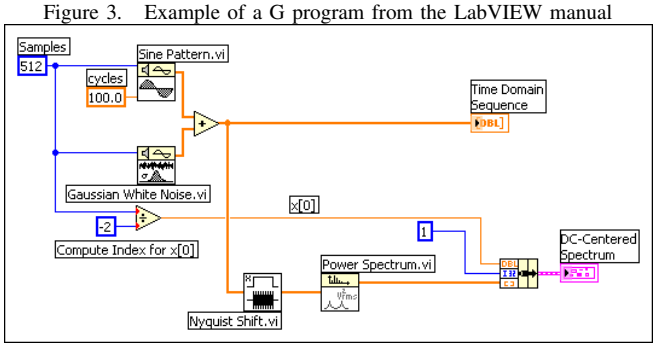
Excel: Spreadsheets are very commonly used in businesses, from inventory administration to educational applications and from scientific modeling to financial systems. Winston [13] estimates that 90% of all analysts in industry perform calculations in spreadsheets. Especially in the financial domain, spreadsheets are the way to perform modeling and programming. Panko [14] estimates that 95% of U.S. firms, and 80% in Europe, use spreadsheets in some form for financial reporting.

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*. A *workbook* consist of a collection of worksheets. Formulas can reference other cells in the same or in a different worksheet. References to a different workbook are possible but less common.

Yahoo! Pipes: A web mashup is a program that collects and combines information from various sources. In Yahoo! Pipes, the information comes from RSS feeds. Figure 2 shows an example Yahoo! Pipes program with 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*. Abstraction is possible in Yahoo! Pipes through the use of a *subpipe* module, which allows a programmer to insert a different pipe as a subroutine, appearing like a standard module with a link to the source.

LabVIEW: LabVIEW is a proprietary, visual hardware system design environment that features the visual programming language "G" as one of its primary features. An example of a G program can be found in Figure 3. G is a dataflow language which means a program is represented as a directed graph where data "flows" between nodes through their edges. As such the two most important primitives in G are the edges called *wires* and nodes called *virtual instruments*. VI's are very similar to functions, because they perform an operation on inputs and provide outputs. Wires

Figure 3. Example of a G program from the LabVIEW manual



with no source VI are inputs and wires without a destination VI are outputs.

III. Smells in End-User Programs

TODO: We need to decide if we refer to smell names With Camel Case or lower case and be consistent. Research into end-user programming 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 [9], and transform them to be applicable to the end-user environment [4]–[6], [11], [15]. 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 [6], by looking at user reports like forum or newsgroup posts [6], [16], or by analyzing publicly available artifacts [4], [11].

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.

A. OO Smells in End-User Programs

We summarize the OO smells present in the three end-user languages, Excel spreadsheets, Yahoo! Pipes mashups, and LabVIEW designs, in Table I.

Overall, we observe a lot of similarities in the code smells studied. For example, the *Duplicate Code* and *Many Parameters* smells have been studied in all three languages. Some smells, like the *Long Method* smell, have been studied in only one domain but are likely applicable in other domains, too (✓*). These smells present opportunities for future research, on which we will elaborate in subsection III-C.

1) Excel: Hermans et al. [5] [15] analogize a workbook to a program, a worksheet to a class inside that program and a cell to a method. Working from this analogy, the classic Fowler smells can be divided into class and method smells. The class smells correspond to *inter-sheet* smells and method smells correspond to *intra-sheet* or *intra-cell* smells.

Hermans et al. [5] define four inter-worksheet smells and define and evaluate threshold for them based on the EUSES spreadsheet corpus [17]. Fowler's class smells are translated into spreadsheets using the above analogy. The *Feature Envy* smell for example indicates that a class uses members of another class more than it uses the members of its own class. Similarly a cell can use cells of other worksheets more than it uses those of its own sheet, and thus that cell could better be placed in the other sheet.

In [15], Hermans et al. define five intra-worksheet or formula smells and again define and evaluate thresholds for them based on the EUSES corpus. Several method-level smells translate well into cell-level smells, supporting the validity of the above analogy. *Multiple Calculations* is similar to the *Long method* code smell, for example, a method with many lines or operations can be hard to understand or change, a cell that has a long formula can have the same problem. Another interesting smell is the *Long Calculation Chain*, which occurs because referenced formulas can reference other formulas ad infinitum. This is also very similar to Fowler's *Message Chain* smell.

2) Yahoo! Pipes: Stolee and Elbaum [4], [11] 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. Another common smell, *Duplicate Modules*, impacted nearly one-quarter of the pipes studied. Again, this is an instance of the *Duplicate Code* smell.

The *Noisy Module* smell impacted 28% of the pipes studied, and it maps to Fowler's *Unused Field* smell. Here, empty or duplicate fields were identified in the pipes, akin to parameters in a method. Overall, 81% of the programs studied from the Yahoo! Pipes community had at least one smell.

3) LabVIEW: Chambers and Scaffidi [6] are primarily interested in smells with potential performance impacts. The only one that has a direct OO equivalent is the *Too Many Variables* smell, which is analogous to the *Many Paramaters* smells of a OO method. **TODO: This explanation does not match the table**

B. Domain-Specific Smells

The end-user programming environments offer many opportunities to define smells based on user behavior or unique elements of the domain. For example, access to large repositories of programs can lead to smells that deviate from best practices in programming. In this section, we explore opportunities for new smells in end-user domains that extend beyond the OO-inspired smells described in Section III-A.

1) Excel: Similar to relational databases, data in spreadsheet is stored in rows and columns. To process data, a

Table I
CODE SMELLS IN END-USER PROGRAMS

OO smell	Excel	Yahoo! Pipes	LabVIEW
Feature Envy	Feature Envy [5]	Feature Envy *	
Long Method	Multiple operations [15]	Large Pipe *	Large Virtual Instrument *
Message Chain	Long Calculation Chain [15]		
Inappropriate Intimacy	Inappropriate Intimacy [5]	Inappropriate Intimacy *	
Lazy class or Middle Man	Middle Man [5]	Noisy Module [11]	
Many Parameters	Multiple References [15]	Many Fields **TODO: correct?** [11]	Many Control Terminals *
Duplicate Code	Duplicated Formulas [15]	Isomorphic Path [11]	Isomorphic Path *
Dead Code	✗	Disconnected or Dangling Modules [11]	Disconnected or Dangling Elements *
Unused Field	✗	✓ [11]	
No-op	Redundant Operations *	Lazy Module [11]	Redundant Operations [6]
Use of Deprecated Interfaces	Deprecated Functions *	Deprecated Modules [11]	

* : Proposed name, future opportunity not supported by prior work.
✗ : Not applicable due to the nature of the paradigm
TODO: I (David) do not fully understand the difference between duplicate module and isomorphic path, so that might be wrong
TODO: Katie, you marked a row but I cannot directly find the mapping, can you fill them in?

formula is placed at the end of this row and copied into the same column of the other rows. 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, Microsoft Excel already detects this smell and shows a small yellow exclamation mark to the user overlaid on the smelly cell.

Another smell specific to spreadsheets is when a formula references an empty cell, which is often in error [18]. This is comparable to a null pointer in a program or a null value in a database, but because a spreadsheet contains both the data and logic we can mark it as a smell.

2) Yahoo! Pipes: By exploring a large subset of the Yahoo! Pipes repository, Stolee and Elbaum identified two smells based on the presence of broken data sources and the use of deprecated modules [11], both of which could apply to professional languages.

A reference to a broken data sources in Yahoo! Pipes is similar to referencing an empty cell in a spreadsheet. In Java, this would manifest as an error with a `FileNotFound` exception at runtime. Such exceptions are not in the Yahoo! Pipes language, causing it to be marked as a smell. Deprecated modules can lead to unexpected behavior. This is similar to annotating Java applications when a new library version is released [19] (of course, transforming the deprecated module is discussed in refactoring, Section IV-B2).

Exploration of the publicly available repositories of code led Stolee and Elbaum. to identify common programming practices, marking deviations from those practices as smells. This is similar to research for professional languages that identifies smells as anti-patterns of poor programming practices (e.g., in UML [20], **TODO: finish this list**).

3) LabVIEW: G programs are often run in an embedded or real-time environment, and as such are very susceptible to performance problems. This motivated Chambers and Scaffidi [6] to G smells with potential performance implications and implement heuristics to identify these smells.

An example of this is the "*Sequence instead of State machine*" smell which is a smell because state machines can be much better optimized by the G compiler.

Because G program heavily rely on concurrency it is also susceptible to problems which occur in concurrent programs. This is the reason for smells like "*Non-reentrant VI*", similar to a non-reentrant method in OO-languages, because this incurs a performance penalty because of the required blocking.

C. Future Opportunities for Smell Detection

There are several OO code smells in Table 2 that apply to all three domains, such as *Duplicate Code*. However, there are other smells that could apply to additional domains, but have not been studied in prior work. Here, we discuss the potential of generalizing some of the OO and domain-specific smell definitions to additional domains.

1) Excel: The *lazy class* smell is a smell that has not been explored yet within the spreadsheet domain. It could be mapped following the translation that was used before: by translating classes to worksheets. Hence a lazy class becomes a lazy worksheet, one with no or little formulas, and no links to other worksheets.

TODO: I (David) have merged Lazy class and middle man, so this wouldn't be true anymore, Middle man is a special case of lazy class, and also I feel almost all lazy sheets would be middle mans. What do you think?

2) Yahoo! Pipes: The *feature envy* smell could also apply when introducing abstraction to Yahoo! Pipes programs. For example, if a pipe has several instances of the same subpipe module, this could be related to excessive use of another class and be smelly.

When a program uses too much abstraction relative to the size of the pipe, for example, if it is composed of only a subpipe and output module, a pipe 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 [11].

The *Long method* smell could apply to Yahoo! Pipes 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 *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.

3) LabVIEW: Research into smells for LabVIEW has focused mostly on performance problems, while traditionally research into code smells and refactorings has focused on maintainability and code quality.

As such inspiration could be drawn for smells that do not necessarily have an impact on performance. We have defined some OO smells that have parallels in LabVIEW.

A *Large Virtual Instrument* would have similar problems to a *Long Method* and could be divided, and like *Many Parameters* make a method hard to understand *Many Control Terminals* could increase the difficulty of understanding a VI. If a block diagram contains a combination of nodes wired identically multiple times, they are *Duplicated Nodes* and should be extracted into their own VI.

IV. Refactoring end-user programs

A way to solve smells is by refactoring, changing the composition of an artifact so that it no longer contains the smell without changing its behavior or output. Now that we have examined smells, we can explore refactoring end-user programs.

As with the smells, many refactorings are inspired by the OO domain, while others are specific to the end-user domain.

A. OO Refactorings

Table **??** lists the OO refactorings adapted for the end-user domains, identifying those that have been studied (✓) and likely apply (✓*). Some of these come from Fowler's definitions while others are from past refactoring research.

1) Excel: Hermans et al. [21] define refactorings corresponding to most of their smells, but most do not have a direct OO equivalent. One that was defined earlier by Badame and Dig [16] is the direct equivalent of *Extract method* and is called *Extract Row or Column* by Badame and *Extra subformula* by Hermans. Because Hermans equates formulas to methods, refactorings that change the parameters of a method also have a direct equivalent in spreadsheet by changing the references in a formula. For example *Remove Parameter* becomes *Remove reference*.

2) Yahoo! Pipes: The refactorings studied in Yahoo! Pipes aim to make pipes smaller, less complex, more maintainable, and easier to understand [11]. Some of the refactorings translated easily from the OO domain. For example, removing dead code was as simple as *removing non-contributing modules* or removing fields in *clean up module* [11]. The *Pull up method* refactoring involved extracting a connected set of modules into into a subpipe and thus is identical to the *Extract Method* refactoring. Once the subpipe was created, all parts of the program with the same pattern of connected modules were replaced with the subpipe module.

3) LabVIEW: While Chambers and Scaffidi have done some work on refactoring [22], few of their refactorings have a direct OO equivalent. *Remove redundant operation* maps to G by removing a no-op.

Sui et. al [23] have done some work on refactoring visual dataflow languages and defined some refactorings which are applicable to G. Most notable is the *Extract sub graph into Node* refactoring which is similar to the *Extract Method* refactoring and moves part of a graph into a separate node which is then included in the original graph. In LabVIEW this could be named *Extract Virtual Instrument*. However, Sui et. al did not implement their proposed refactorings in an end-user programming environment.

B. Domain-Specific Refactorings

Building off the discussion of domain-specific smells in Section III-B, here we discuss opportunities for domain-specific refactorings.

1) Excel: Some work has been done on refactoring formulas inside a single cell of a spreadsheet. Badame and Dig [16] define the *Guard Call* refactoring, which places a conditional check around a (sub)formula that can return an error, for example a check for a division by zero.

Hermans and Dig [12] have followed up on this and define a more general approach by providing a way to transform formulas into other formulas, in a way that is very similar to how a regular expression or patterns works in some modern programming languages. To define a transformation rule one defines a formula to match and a formula to replace it with. While every formula can be provided, the power comes from, to use the regular expression terminology, "capturing

Table II
CODE REFACTORING IN END-USER PROGRAMS

OO smell	Excel	Yahoo! Pipes	LabVIEW
Remove Parameter	Move References [21]	Remove Field [11]	Remove Terminal [22]
Extract Method	Extract Subformula [16], [21]	Extract Local Subpipe [11]	Extract Virtual Instrument [23]
Inline Method	Inline Formula *	Inline Module [11]	Inline Virtual Instrument *
Substitute Algorithm	Replace Formula *	✓ [11]	Substitute Block Diagram *
Library Migration [24]	Migrate Formulas [12]	Replace Deprecated Modules [11]	
Define named constant	Extract Literal [16]	Pull Up Module [11]	
Remove dead code	✗	Remove Disconnected or Dangling Modules [11]	Remove Disconnected or Dangling Elements *
Remove No-op	Remove Redundant Operations *	Remove Lazy Module [11]	Remove Redundant Operations [22]

* : Proposed name, future opportunity not supported by prior work.

✗ : Not applicable due to the nature of the paradigm

TODO: Merged Extract Method, Pull Up Method and Form Template Method because they are similar in OO so the differences are lost in the translations, ok?
TODO: I don't know how I feel about the substitute algorithm refactoring. It always seems corny/bland to me, because it basically is "replace something bad with something better", at least when you translate it into other paradigms. Should we leave it in?

groups" which can bind to a subformula, cell or range. For example, the transformation rule

$$F_1/F_2 \leftrightarrow IF(F_2 <> 0, F_1/F_2, ''undefined'')$$

implements the previously mentioned *Guard Call* [16] refactoring by transforming a formula into one which prevents division by zero. **TODO: Should we remove the transformation rule discussion? Seems a bit too technical for the tone of the rest of the paper.** In this example F can capture (bind to) any valid formula. With this approach it is possible to create a rule for all refactorings defined by Badame and Dig [16], but also all refactorings as long as they do not require more intricate capturing groups.

2) Yahoo! Pipes: Based on the smell of *non-conforming module orderings* where a programming syntax was different than the community standard, the refactoring, *normalize order of operations* was introduced. This refactoring was shown to preferable to end-user programers for improving the understandability of the program [11]. The presence of a deprecated module creates an opportunity to replace it, similar to updating legacy code in the presence of new library versions (e.g., [19], [24]).

3) LabVIEW: Chambers and Scaffidi [22] have defined transformations for most of their performance smells, but not all are refactorings because some change the program behavior and the ones that did not modify program behavior turned out to not be particularly effective for this goal.

However, some refactorings which only slightly change program behavior were effective. For example introducing a 1ms pause inside a loop without a pause decreased execution time by 37%.

C. Future Opportunities for Refactoring

Future opportunities for refactoring research abound in these domains. Here, we discuss a few opportunities.

1) Excel: While some work has been done on refactoring the structure of the whole spreadsheet instead of only formulas in cells, there is an opportunity for more research. Specifically how worksheets and regions of cells are similar and different to classes and methods can be further explored to define refactorings. A refactoring which was not previously defined is the *Inline Formula* refactoring, which is done by replacing the reference to a cell by its formula and deleting the original cell, although this might not be possible if the cell is referenced as part of a range. The *Remove Redundant Operations* refactoring could be adapted from other domains and is defined as removing parts of a formula which do not impact the result.

Another domain to draw inspiration from is (relational) databases. As spreadsheets are often used to store data and lookup functions are similar to the `JOIN` functionality in SQL, spreadsheets might benefit from knowledge gained in database theory, for example normalization.

Inspiration might also be drawn from refactoring in dataflow languages like Yahoo! Pipes and LabVIEW G, because a spreadsheet can also be represented as a dataflow program in which cells are nodes and reference to cells are wires.

2) Yahoo! Pipes: Handling the smells identified in future work in Section III-C2 involves small modifications on existing refactorings, new refactorings, or may not be possible. For a small modification, the *long method* smell can be addressed with *pull up method*. With *inappropriate intimacy*, a new refactoring should be introduced that is the reverse of *extract local subpipe*, which would, in essence, inline

the subpipe logic. Addressing *feature envy* may be possible in some cases by using *collapse duplicate path* when the subpipes are connected to modules that form similar paths in the pipe. However, in the general case, refactoring this smell may not be feasible.

In a qualitative analysis of the end-user preferences for Yahoo! Pipes programs, the participant responses showed preference toward programs with familiar elements [25]. This opens an opportunity and perhaps a need to define code smells and refactorings based on end-user experience.

3) LabVIEW: Similarly to the research into smells in LabVIEW G programs, research into refactoring of G programs has also been driven by the desire for performance improvements. Refactorings which do not directly impact performance have only been theoretically explored for general dataflow languages [23].

V. Related Work

This paper builds upon the extensive body of work related to code smells and refactoring. For an extensive overview we review the reader to the work of Mens and Tourwé [26].

In end-user programming environments user input and logic are often more closely linked than they are in general purpose languages. As such analyzing the input data as opposed to the logic can also be used to detect problems.

Cunha et. al [18] claim to define code smells for spreadsheet, but for most of their smells they look at different types of anomalies in the data of a spreadsheet and define these as smells. "Data Smell" might be a more appropriate name for such 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 [27] take a very similar but more formalized approach which they label "Data Debugging". They developed an Excel plugin called CheckCell which 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.

VI. Discussion

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

A. Future Opportunities in Other EUP Domains

End-user programming domains extend beyond spreadsheets, web mashups, and system designs. Stolee and Elbaum explore future opportunities for refactoring in educational programming languages (i.e., Scratch [28], Alice [29], or Kodu [30].) and web macros (e.g., CoScripter [31]). Additional opportunities in educational languages exist in LEGO Mindstorms [32], which is based off the G language for LabVIEW.

Another end-user programming domain that could benefit from smell analysis and refactoring is mathematics languages. These include matlab, sage, or mathematica. **TODO: more here**.

B. Future Opportunities in Professional Languages

In end-user programming languages, it has been shown that code smells impact the understandability of source code [11]. Additionally, being presented with code smells can motivate end-user programmers to improve their code [6], and smells in spreadsheets have even been known to reveal actual errors [15]. These lessons could extend to other programming languages outside of the end-user programming domains. There has been successful in using automating smell detection, for example, during agile development (e.g., [33]). Paired with the end-user evidence, a stronger case can be made to integrate automated smell detection in many domains.

C. Threats to Validity

The threats to validity of this work inherit the threats to validity of the original studies [3]–[6], [11], [12], [15], [16], [22], [25]. In addition, we note that the authors of this work have pioneered smell detection and refactoring research in spreadsheets and Yahoo! Pipes, but are not involved with LabVIEW. Thus, the opportunities for future work in this area may not be complete.

VII. Concluding Remarks

REFERENCES

[1] C. Scaffidi, M. Shaw, and B. A. Myers, "Estimating the numbers of end users and end user programmers," in *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2005)*, 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 Comput. Surv.*, vol. 43, no. 3, pp. 21:1–21:44, Apr. 2011. [Online]. Available: http://doi.acm.org/10.1145/1922649.1922658

[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] K. Stolee and S. Elbaum, "Refactoring pipe-like mashups for end-user programmers," in *33rd International Conference on Software Engineering*, 2011. [Online]. Available: http://www.neverworkintheory.org/?p=13

[5] 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.

[6] C. Chambers and C. Scaffidi, "Smell-driven performance analysis for end-user programmers," in *Visual Languages and Human-Centric Computing (VL/HCC), 2013 IEEE Symposium on*. IEEE, 2013, pp. 159–166.

[7] W. F. Opdyke, "Refactoring object-oriented frameworks," Ph.D. dissertation, Champaign, IL, USA, 1992, uMI Order No. GAX93-05645.

[8] W. G. Griswold and D. Notkin, "Automated assistance for program restructuring," *ACM Trans. Softw. Eng. Methodol.*, vol. 2, no. 3, pp. 228–269, Jul. 1993. [Online]. Available: http://doi.acm.org/10.1145/152388.152389

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

[10] T. Mens and T. Tourwé, "A survey of software refactoring," *IEEE Trans. Softw. Eng.*, vol. 30, no. 2, pp. 126–139, Feb. 2004. [Online]. Available: http://dx.doi.org/10.1109/TSE.2004.1265817

[11] K. T. Stolee and S. Elbaum, "Identification, impact, and refactoring of smells in pipe-like web mashups," *IEEE Transactions on Software Engineering*, vol. 39, no. 12, pp. 1654–1679, 2013.

[12] F. Hermans and D. Dig, "Bumblebee: a refactoring environment for spreadsheet formulas," in *Proceedings of the International Symposium on Foundations of Software Engineering '14*, 2014, pp. 747–750.

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

[14] R. Panko, "Facing the problem of spreadsheet errors," *Decision Line*, vol. 37, no. 5, 2006.

[15] F. Hermans, M. Pinzger, and A. van Deursen, "Detecting code smells in spreadsheet formulas," in *Proc. of ICSM '12*, 2012.

[16] S. Badame and D. Dig, "Refactoring meets spreadsheet formulas," in *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. IEEE, 2012, pp. 399–409.

[17] M. Fisher and G. Rothermel, "The euses spreadsheet corpus: a shared resource for supporting experimentation with spreadsheet dependability mechanisms," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4. ACM, 2005, pp. 1–5.

[18] J. Cunha, J. P. Fernandes, H. Ribeiro, and J. Saraiva, "Towards a catalog of spreadsheet smells," in *Computational Science and Its Applications–ICCSA 2012*. Springer, 2012, pp. 202–216.

[19] W. Tansey and E. Tilevich, "Annotation refactoring: Inferring upgrade transformations for legacy applications," *SIGPLAN Not.*, vol. 43, no. 10, pp. 295–312, Oct. 2008. [Online]. Available: http://doi.acm.org/10.1145/1449955.1449788

[20] T. v. Enckevort, "Refactoring uml models: Using openarchitectureware to measure uml model quality and perform pattern matching on uml models with ocl queries," in *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '09. New York, NY, USA: ACM, 2009, pp. 635–646. [Online]. Available: http://doi.acm.org/10.1145/1639950.1639959

[21] F. Hermans, M. Pinzger, and A. van Deursen, "Detecting and refactoring code smells in spreadsheet formulas," *Empirical Software Engineering*, pp. 1–27, 2014. [Online]. Available: http://dx.doi.org/10.1007/s10664-013-9296-2

[22] 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.

[23] Y. Y. Sui, J. Lin, and X. T. Zhang, "An automated refactoring tool for dataflow visual programming language," *ACM Sigplan Notices*, vol. 43, no. 4, pp. 21–28, 2008.

[24] I. Balaban, F. Tip, and R. Fuhrer, "Refactoring support for class library migration," *SIGPLAN Not.*, vol. 40, no. 10, pp. 265–279, Oct. 2005. [Online]. Available: http://doi.acm.org/10.1145/1103845.1094832

[25] K. T. Stolee, J. Saylor, and T. Lund, "Exploring the benefits of using redundant responses in crowdsourced evaluations," in *Proceedings of the 2nd International Workshop on Crowd-Sourcing in Software Engineering*, ser. CSI-SE 2015, 2015, to appear.

[26] T. Mens and T. Tourwé, "A survey of software refactoring," *Software Engineering, IEEE Transactions on*, vol. 30, no. 2, pp. 126–139, 2004.

[27] D. W. Barowy, D. Gochev, and E. D. Berger, "Checkcell: data debugging for spreadsheets," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*. ACM, 2014, pp. 507–523.

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

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

[30] 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.

[31] G. Leshed, E. M. Haber, T. Matthews, and T. Lau, "Co-scripter: automating & sharing how-to knowledge in the enterprise," in *Proceedings of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*, ser. CHI '08, 2008, pp. 1719–1728.

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

[33] J. Schumacher, N. Zazworka, F. Shull, C. Seaman, and M. Shaw, "Building empirical support for automated code smell detection," in *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '10. New York, NY, USA: ACM, 2010, pp. 8:1–8:10. [Online]. Available: http://doi.acm.org/10.1145/1852786.1852797