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 to design hardware circuits, combine web information, and impact business decisions. Software engineering research into refactoring has traditionally focused on professionally used 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 design 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 on refactoring 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 end-user programming domains.

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

I. Introduction

End-user programmers are said to outnumber the amount of professional programmers three times over [1]. These 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, 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*.

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 [6]–[8]. Alleviating those smells can be achieved with refactoring.

Refactoring was first introduced as a systematic way to restructure source code and facilitate software evolution and maintenance. Martin Fowler later introduced the concept of code smells [9]. Refactoring code is often motivated by noticing a code smell, which signals the opportunity for improvement.

The taxonomy of smells outlined in Fowler's text pertained mostly to object-oriented code, and professional programming languages were the 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 paradigms, including web mashups [4], [8], Excel spreadsheets [3], [5], [11], and LabVIEW programs [6], all of which are end-user programming domains.

Considering the large number of end-user programmers, the longevity of their artifacts, and the impact of smells on understandability, errors, and performance, supporting end-user programmers in code smell detection and refactoring is valuable. 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 and refactorings 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:

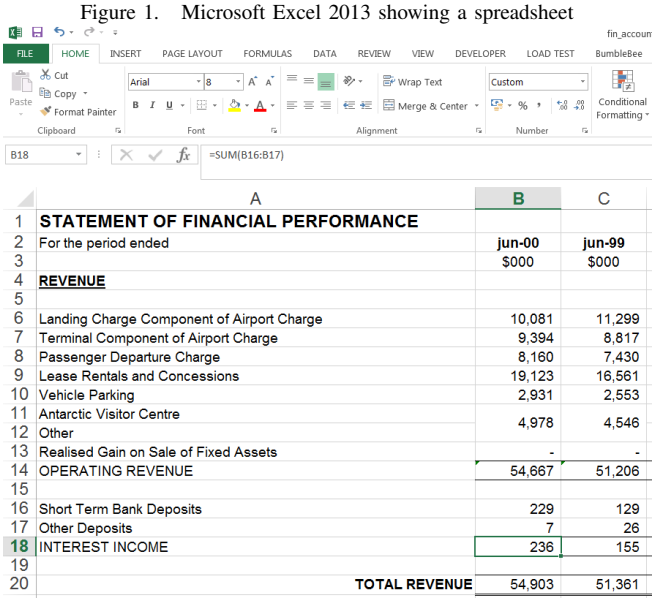
- Synthesis and catalog of object-oriented-inspired code smells and refactoring in end-user programs
- Discussion of how smells and refactoring in end-user domains may translate to professional languages
- Identification of future opportunities for smell detection and refactoring in end-user programming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

II. Background

Here, we briefly explore each end user domain 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 [12] estimates that 90% of all analysts in industry perform calculations in spreadsheets. Microsoft Excel is by

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

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*. A *workbook* consist of a collection of worksheets. Formulas can reference other cells in the same or in different workbooks and worksheets.

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. 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.

LabVIEW: LabVIEW is a 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 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

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*.

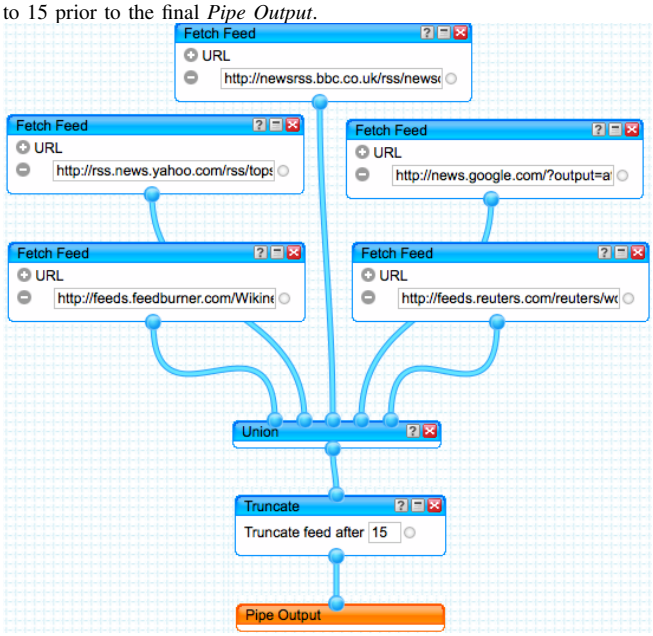
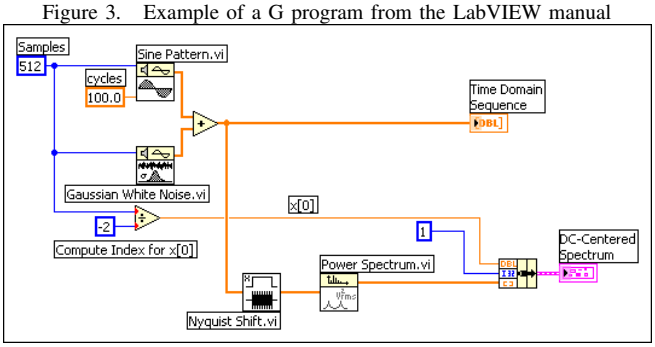


Figure 3. Example of a G program from the LabVIEW manual



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]–[8]. 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], [13], or by analyzing publicly available artifacts [4], [8].

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.

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 [7]	Long Module*	Large Virtual Instrument *
Message Chain	Long Calculation Chain [7]		
Inappropriate Intimacy	Inappropriate Intimacy [5]	Inappropriate Intimacy *	
Lazy class or Middle Man	Middle Man [5]	Unnecessary Abstraction [8]	
Many Parameters	Multiple References [7]		Many Control Terminals *
Duplicate Code	Duplicated Formulas [7]	Duplicate Modules, Duplicate String or Isomorphic Path [8]	Isomorphic Path *
Dead Code	✗	Disconnected or Dangling Modules [8]	Disconnected or Dangling Elements *
Unused Field	✗	Noisy Module [8]	
No-op	Redundant Operations *	Unnecessary Module [8]	Redundant Operations [6]
Use of Deprecated Interfaces	Deprecated Functions *	Deprecated Module or Invalid Source [8]	

✗ : Not applicable due to the nature of the paradigm
* : Proposed name, likely future opportunity not supported by prior work
⟨blank⟩ : Not discussed in this work, possible future opportunity

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 (marked with *). These smells present opportunities for future research (see Section III-C).

1) Excel: **TODO: trim down this section. Katie thinks we should favor the domain-specific and future work discussions over the prior work** Hermans et al. [5] [7] 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 [14]. 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 [7], 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], [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 the pipes studied. 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 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 [15]. 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 [8]. A reference to a broken data sources 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 [16] (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 identifying smells as anti-patterns and could be extended to any language.

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. 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.

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.

Table II
CODE REFACTORINGS IN END-USER DEVELOPMENT

OO Refactoring	Excel	Yahoo! Pipes	LabVIEW
Remove Parameter	Move References [17]	Clean Up Module [8]	Remove Terminal [18]
Extract Method	Extract Subformula [13], [17]	Extract Local Subpipe [8]	Extract Virtual Instrument [19]
Inline Method	Inline Formula *	Push Down Module [8]	Inline Virtual Instrument *
Substitute Algorithm	Replace Formula *	Merge Redundant Modules or Collapse Duplicate Path [8]	Substitute Block Diagram *
Library Migration [16]	Migrate Formulas [11]	Replace Deprecated Modules [8]	
Define named constant	Extract Literal [13]	Pull Up Module [8]	
Remove dead code	✗	Remove Disconnected or Dangling Modules [8]	Remove Disconnected or Dangling Elements *
Remove No-op	Remove Redundant Operations *	Remove Lazy Module [8]	Remove Redundant Operations [18]

✗ : Not applicable due to the nature of the paradigm
* : Proposed name, future opportunity not supported by prior work.
⟨blank⟩ : Possible future opportunity, not discussed in this work

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. 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 II 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. [17] 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 [13] 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 [8]. Some of the refactorings translated easily from the OO domain. For example, removing dead code was as simple as *removing disconnected, dangling, or swaying modules* or removing parameters in *clean up module* [8]. The *Extract Local Subpipe* 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 [18], few of their refactorings have a direct OO equivalent. *Remove redundant operation* maps to G by removing a no-op.

Sui et. al [19] 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 [13] 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 [11] 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.

With this approach it is possible to create a rule for all in-formula refactorings defined by Badame and Dig [13], but also other 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 [8]. 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., [16]).

3) LabVIEW: Chambers and Scaffidi [18] 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 or new refactorings. For a small modification, the *long module* smell can be addressed with *extract local subpipe*. 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. With *inappropriate intimacy*, a new refactoring should be introduced that is the reverse of *extract local subpipe*, which would inline the subpipe logic.

In a qualitative analysis of the end-user preferences for Yahoo! Pipes programs, the participant responses showed preference toward programs with familiar elements [20]. 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 [19].

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 [21].

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 [15] 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. In more recent work, Barowy et. Al [22] 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 and other end-user domains.

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 [8]. Additional opportunities in educational languages exist in LEGO Mindstorms, 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.

In particular, the smells related to duplication and poor construction (e.g., long method, many parameters, dead code) are prevalent in the three domains studied. These smells – and their respective refactorings – likely exist

in other end-user programming domains, and likely hinder the understandability and maintainability of those programs. Worse even, these smells likely also lead to errors, and thus are worthy of our attention.

B. Threats to Validity

The threats to validity of this work inherit the threats to validity of the original studies [3]–[8], [11], [13], [18], [20]. 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

This paper presents an overview of the work in smell detection and refactoring for end-user programming languages. More specifically, it synthesizes work on Yahoo! Pipes, Excel and LabView. We explore commonalities between these works and identify opportunities for application of methods from these domains to others and to generic programming languages. As we move forward, we see many opportunities to explore additional smells and refactorings in the domains studied, in other end-user programming domains, and even in extending the findings to professional languages.

Acknowledgements

This work is supported in part by NSF SHF-EAGER-1446932 and the Harpole-Pentair endowment at Iowa State University. **TODO: Felienne - do you have funding agencies to thank?**

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

[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 *Proc. of VLH/CC '13*, 2013, pp. 159–166.

[7] F. Hermans, M. Pinzger, and A. van Deursen, "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. on SE*, vol. 39, no. 12, pp. 1654–1679, 2013.

[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. on SE*, vol. 30, no. 2, pp. 126–139, Feb. 2004.

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

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

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

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

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

[16] I. Balaban, F. Tip, and R. Fuhrer, "Refactoring support for class library migration," *ACM SIGPLAN Notices*, vol. 40, no. 10, pp. 265–279, Oct. 2005.

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

[18] 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.

[19] 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.

[20] 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.

[21] T. Mens and T. Tourwé, "A survey of software refactoring," *IEEE Trans. on SE*, vol. 30, no. 2, pp. 126–139, 2004.

[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.