Smell Detection and Refactoring in End-User Programming Languages

David Hoepelman
Delft University of Technology
Delft, The Netherlands
D.J.Hoepelman@student.tudelft.nl

Kathryn T. Stolee
Iowa State University
Ames, IA, USA
kstolee@iastate.edu

Felienne Hermans
Delft University of Technology
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 write code to design hardware circuits, combine web information, and make 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 3) identify future opportunities for smell detection 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 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 to object-oriented (OO) 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 languages.

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. The contributions of this work are:

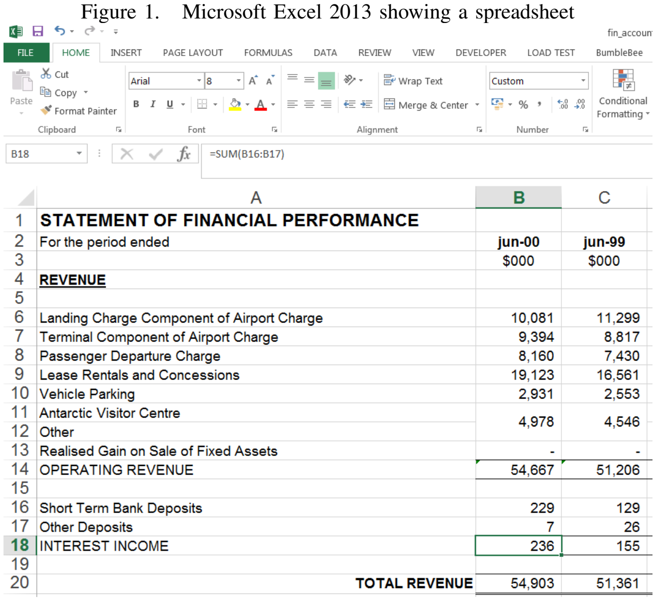
- Synthesis and catalog of object-oriented-inspired code smells and refactoring in end-user programs
- Discussion of unique smell detection and refactoring opportunities in the end-user domains
- Identification of future opportunities for smell detection and refactoring in end-user programming domains

II. Background

Here, we briefly explore each end-user language targeted by prior refactoring research 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 far the most used, and therefore most studied, spreadsheet program, but other implementations exists and are similar.

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. 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*.

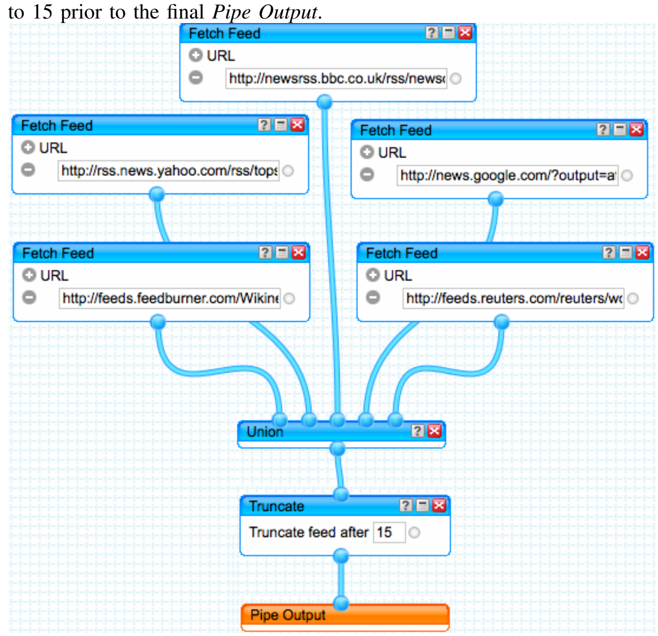
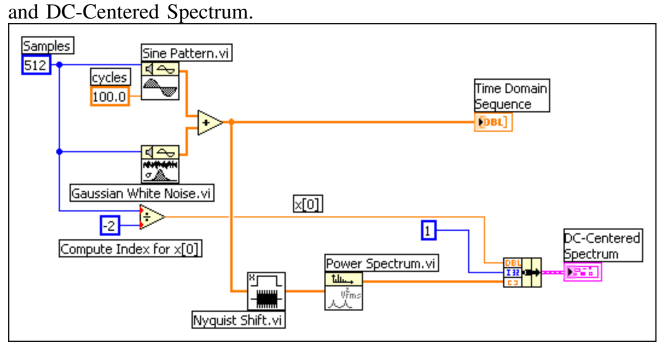


Figure 3. Example of a G program from the LabVIEW manual. This program takes samples from sensors and show their Time Domain Sequence and DC-Centered Spectrum.



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: A web mashup is a program that collects and combines information from various web data sources. In Yahoo! Pipes, the information comes from RSS feeds. 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.

LabVIEW: LabVIEW is a hardware system design environment that features the visual programming language "G" . An example of a G program can be found in Figure 3. The G language is a visual, dataflow language where data "flows" between nodes through their edges. The two most important primitives in G are the edges (*wires*) and nodes (*virtual instruments* (VI)). Virtual instruments are very similar to functions, performing operations on inputs and providing outputs. Wires with no source VI are inputs and wires without a destination VI are outputs.

III. 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 [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.

A. OO Smells in End-User Programs

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

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 Paths [8]	Isomorphic Paths *
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

Overall, we observe a lot of similarities in the code smells studied. For example, the *Duplicate Code* smell has been studied in two of the 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: 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 and use this to transform nine of Fowlers smells. An example of this is the *Long Method* smells, which translated the *Multiple Operations* smells because formulas with a lot of operations are long and suffer from similar problems as long methods.

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.

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

3) LabVIEW: Current work on LabVIEW smells has been focused on smells with potential performance impacts. As most OO smells do not have a direct impact on performance, the equivalents of OO smells for LabVIEW have not yet been defined. *Redundant Operations* is an exception, and is similar to a statement with no effect, a *No-op*.

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 in Section III-A.

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.

Another smell specific to spreadsheets is when a formula references an empty cell, which is often in error [14]. This is comparable to a null pointer in a program, but because a spreadsheet contains both the input 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 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. In Java, this would manifest as an error with a `FileNotFound` exception at runtime. 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.

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 the G compiler can better optimize state machines.

Since G programs heavily rely on concurrency, they are susceptible to concurrency-related problems programs such as non-reentrant VI's.

C. Future Opportunities for Smell Detection

There are several OO code smells in Table 2 that could apply to all three domains, such as *Duplicate Code*, bur prior work in LabVIEW does not cover it. Here, we discuss the potential of generalizing some of the OO and domain-specific smell definitions to additional domains.

1) Excel: Most of the smells studied in other end-user domains have been studied in Excel spreadsheets, but there is some future potential in the areas of redundancy and deprecation.

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

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.

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 identified 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, these are *Isomorphic Paths* and should be extracted into their own VI. A VI node that is *Disconnected or Dangling* does not contribute to the program and could also be marked as smelly.

IV. Refactoring end-user programs

A way to solve smells is by refactoring, changing 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. [15] 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.

Table II
CODE REFACTORINGS IN END-USER DEVELOPMENT

OO Refactoring	**Excel**	**Yahoo! Pipes**	**LabVIEW**
Remove Parameter	Move References [15]	Clean Up Module [8]	Remove Terminal [16]
Extract Method	Extract Subformula [13], [15]	Extract Local Subpipe [8]	Extract Virtual Instrument [17]
Inline Method	Inline Formula *	Push Down Module [8]	Inline Virtual Instrument *
Substitute Algorithm	Replace Formula *	Merge Redundant Modules or Colapse Duplicate Path [8]	Substitute Block Diagram *
Library Migration [18]	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 [16]

✗ : 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

3) LabVIEW: While Chambers and Scaffidi have done some work on refactoring [16], few of their refactorings have a direct OO equivalent. *Remove Redundant Operation* maps to G as removing a No-op. However, Sui et. al [17] have done some work on refactoring general visual dataflow languages and defined some refactorings which are applicable to G. Most notably, their work defines the *Extract Virtual Instrument* when applied to G.

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 generalized 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.

2) Yahoo! Pipes: When a program used a syntax that was different from the community standard, as detected by the *Non-conforming Module Orderings* smell, the *Normalize Order of Operations* refactoring could be applied to change this. The end-user programmers preferred the standardized version [8].

3) LabVIEW: Chambers and Scaffidi [16] have defined transformations for most of their performance smells, but not all are refactorings because most change the program behavior. 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 are abound in these domains. We will discuss a few of these.

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.

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. 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.

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.

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 [17].

We identify three concrete opportunities based on other domains. First of the counterpart to *Extract Virtual Instrument* is *Inline Virtual Instrument*, which is the refactoring that replaces a VI by its implementation. Secondly the dead code elimination refactorings from the Yahoo! Pipes domain are applicable to LabVIEW as both are dataflow languages. Lastly the equivalent to the *Substitute Algorithm* refactoring would be the *Substitute Block Diagram*

V. Related Work

For an extensive overview of refactoring in professional languages, we refer the reader to the work of Mens and Tourwé [19].

In end-user programming, input data and logic are often more closely linked than they are in general purpose languages. This allows smells to be defined by characteristics of the input data. Cunha et. al [14] define several smells based on user input data. In more recent work, Barowy et. Al [20] take a more formalized approach which they label "Data Debugging", which uses statistical analysis to find values with a high impact on the results of a spreadsheet.

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 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, or in MAX/MSP, which are visual programming languages for music and multimedia. Another end-user programming domain that could benefit from smell analysis and refactoring is mathematics languages. These include MATLAB (e.g., the Simulink language), 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], [16], [21].

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).

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 to other end-user programming domains. 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.

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. Soft. Eng.*, 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. Soft. Eng.*, 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] J. Cunha, J. P. Fernandes *et al.*, “Towards a catalog of spreadsheet smells,” in *Proc. of ICCSA ’12*. Springer, 2012, pp. 202–216.

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

[16] 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.

[17] 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.

[18] 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.

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

[20] 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.

[21] 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.