End-User Refactoring

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*—TODO: Write abstract, what do we want to point out?**

***Keywords*-end-user computing; 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].

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 the 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]. In their lifespan, these artifacts are modified, often by different people. This makes them vulnerable for code *smells*, like other source code artifacts.

These 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, which leads to the topic of this paper: *how can we support end-users in recognizing code smells and refactoring their artifacts?* **TODO: This is up for discussion.** The contributions of this work are:

- Motivation for the study of smells and refactoring for end-user programming languages based on empirical evidence
- Synthesis and catalog of code smells and refactoring in end-user programs
- Identification of future opportunities for smell detection and refactoring in end-user programming domains

II. Background

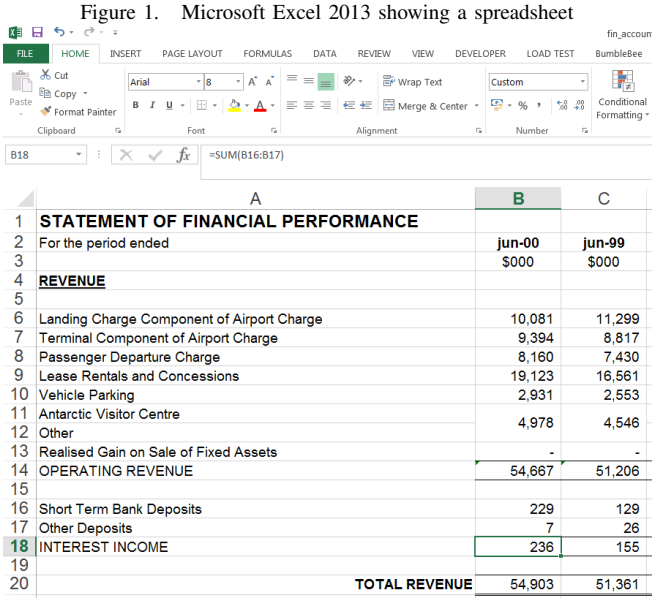
Refactoring was first introduced as a systematic way to restructure source code and facilitate software evolution and maintenance [7], [8]. In 1999, Martin Fowler introduced the concept of code smells [9]. According to Fowler, code smells indicate suspicious or weak parts that the developer might want 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 of refactoring research 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], [11], Excel spreadsheets [3], [5], [12], and dataflow programs [6], all of which are end-user programming domains.

Researchers have also 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 (p=0.01, [11]). **TODO: add a finding about the impact of smells on spreadsheets** When presenting LabVIEW users with information about code smells in their programs, all thirteen tool-users were able to identify the problems in the given programs and twelve were able to fix the problem [6]. Considering the large number of end-user programmers, the longevity of their artifacts, and the impact of smells, supporting end-user programmers in code smell detection and refactoring is important.

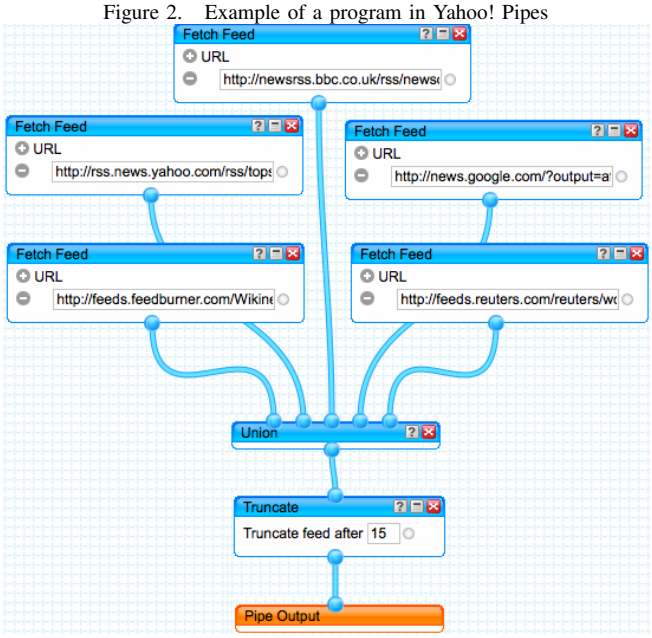
In this work, we explore past research in three end-user programming domains: spreadsheets in Excel, web mashups in Yahoo! Pipes, and and visual system design in LabVIEW, and identify future opportunities for smell and refactoring research in these and other end-user programing domains.

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



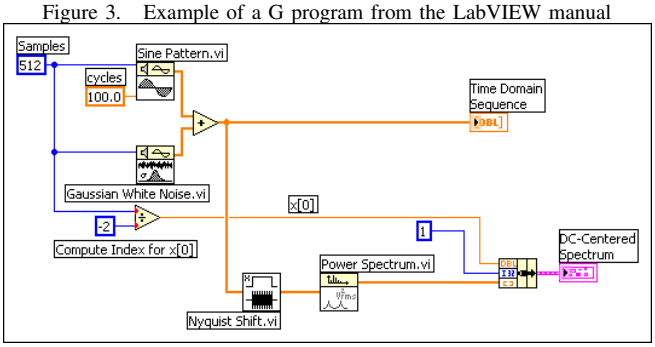
Here, briefly explore each domain before presenting the relevant code smells in Section III.

Excel: **TODO: Fellienne - short intro to spreadsheets.** 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.

LabVIEW: LabVIEW is a proprietary, visual hardware system design environment which 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.

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



III. Smells in End-User Programs

Research into end-user programming smells has had two approaches, which are not mutually exclusive. The first approach is to take existing smells for object-oriented programming languages, usually those defined by Fowler, and transform them to be applicable to the end-user environment [4], [5], [11], [13]. 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], [14], 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.

Table I
OO CODE SMELLS IN END-USER PROGRAMS

Smell	Excel	Yahoo! Pipes	LabVIEW
Feature Envy	✓ [5]	✓*	
Long Method	✓ [13]	✓*	✓*
Message Chain	✓ [13]		✓ [6]
Inappropriate Intimacy	✓ [5]	✓*	
Middle Man	✓ [5]	✓ [11]	
Shotgun Surgery	✓ [5]		
Many Parameters	✓ [13]	✓ [11]	✓ [6]
Duplicate Code	✓ [13]	✓ [11]	✓ [6]
Lazy Class	✓*	✓ [11]	✓ [15]
Dead Code	✗	✓ [11]	✓ [6]
Unused Field	✗	✓ [11]	

TODO: Should we say lazy class or redundant operation? The first is a close OO equivalent, the latter is more appropriate for what we mean in all 3 cases. Stolee calls it lazy class, Chambers redundant operations
TODO: Merge dead code and unused field? They are somewhat similar
TODO: Remove shotgun surgery?
Key:
✗ : Not applicable
✓ : Applicable, supported in prior work
✓* : Likely applicable, 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, and LabVIEW, in Table I.

Overall, we observe a lot of overlap 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 studies in only one domain but are a likely applicable in other domains too (✓*). These and smells for which we do not know if they apply present opportunities for future research, on which we will elaborate in subsection III-C.

A small number of smells is definitely not applicable (✗) because of differences in the domain. Spreadsheets for example cannot contain dead code because the user might still be interested in a piece of data even if it is not used anywhere else in the spreadsheet.

1) Excel: Hermans et al. [5] [13] 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 [16]. Fowler's class smells are translated into spreadsheets using the above analogy. For example the *Feature Envy* smell 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 [13], 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. Like 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 happens because referenced formulas can reference other formulas ad infinitum. This is very similar to Fowler's *Message Chain* smell.

2) Yahoo! Pipes: Stolee et al. [4], [11] treat a Yahoo! Pipes mashup as a program and each module as a method. The most common smell, appearing in nearly one-third of the 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, but at a higher level of abstraction.

The *Noisy Module* smell impacted 28% of the pipes studied, and it maps to Fowler's *Unused Field* smell. Here, we found empty or duplicate fields in the pipes, akin to parameters in a method.

3) LabVIEW: Chambers and Scaffidi [6] are primarily interested in smells with potential performance impacts. The only one which has a direct OO equivalent is the *Too Many Variables* smell, which is analogous to the *Many Paramaters* smells of a OO method.

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.

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 [17]. This is similar 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 et al. identified two smells based on the presence of broken data sources and the use of

deprecated modules [11]. These are based on the presence of unsupported, or deprecated, modules, and trying to access an RSS feed that either does not exist or requires special permission.

The presence of a large, publicly available repositories of code opens an opportunity to identify common programming practices, and deviations from those practices as smells. This approach was used to define smells based on community standards for the Yahoo! Pipes domain.

3) LabVIEW: G programs are often run in an embedded or real-time environment, and as such are very susceptible to performance problems. This has 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 smell definitions to additional domains.

1) Excel: **TODO: Felienne**

All work on spreadsheet smells has been done with the Microsoft Excel implementation, which is by far the most used spreadsheet software. **TODO: cite reference?** However, other spreadsheet software exists. Because these all operate on the same principles we expect all smells to be applicable there too, but this can be confirmed in future work.

2) Yahoo! Pipes: Many of the smells studied in Excel and LabVIEW could apply to Yahoo! Pipes, and in particular, *Feature Envy* and *Inappropriate Intimacy*. 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, this smell matters to end-user programmers. In an empirical evaluation, programmers often preferred pipes without subpipe modules because they were easier to understand [11].

Table II
OO CODE REFACTORING IN END-USER PROGRAMS

Refactoring	Excel	Yahoo! Pipes	LabVIEW
Remove Parameter	✓ [18]	✓ [11]	✓*
Extract Method	✓ [14], [18]	✓*	✓ [20]
Inline Method	✓ [18]	✓ [11]	✓*
Form Template Method		✓ [11]	
Pull Up Method		✓ [11]	
Substitute Algorithm	✓*	✓ [11]	✓*
Library Migration [19]	✓*	✓ [11]	✓*
Define named constant	✓ [14]	✓*	
Remove dead code	✗	✓ [11]	✓*
Remove redundant operations	✓*	✓ [11]	✓ [15]

TODO: How do we handle smells and refactorings which are trivially connected? E.g. "Dead code" and "Remove dead code", "Long method" and "extract method", "Magic number" and "Define named constant". Related, how do we handle refactorings with no smell defined?

Key:
✗ : Not applicable
✓ : Supported in prior work
✓* : Likely applicable, future opportunity

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. While these have some overlap, some smells do not have an impact on performance and as such could be further explored. And example of this is the *Long Method* smell, which in G could be a smell related to large VI's.

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 continue to the refactoring of end-user programs.

As with the smells, many refactorings are inspired by the OO domain, while others are specific to the end-user domain. Note that prior work in LabVIEW does not define refactorings, creating a clear opportunity for future work.

A. OO Refactorings

1) Excel: Hermans et al. [18] define refactorings corresponding to most of their smells, but most do not have a direct OO equivalent. One which was defined earlier by Badame and Dig [14] 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 were target the code smells with the goal of making 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. 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 [15], all of their refactorings have no direct OO equivalent. *Remove redundant operation* is an exception to this, and is simply the removal of a no-op. However, the LabVIEW programming language G is a visual dataflow language and Sui et. al [20] 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

1) Excel: Some work has been done on refactoring formulas inside a single cell of a spreadsheet. Badame and Dig [14] 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 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* [14] refactoring by transforming a formula into one which prevents division by zero. 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 [14], but also all refactorings as long as they do not require more intricate capturing groups.

2) Yahoo! Pipes: Opportunities for domain-specific refactorings stem from the domain-specific code smells. For example, 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].

3) LabVIEW: Chambers and Scaffidi [15] 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 loops without a pause decreased execution time by 37%.

C. Future Opportunities for Refactoring

1) Excel: While some work has been done on refactoring the structure of the whole spreadsheet instead of only formulas in cells, there is possibility 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.

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: **TODO: Katie**

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 lanuages [20].

V. Related Work

A. "Data smells"

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

Cunha et. al [17] 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. As such "Data Smell" is a more appropriate name. Examples of this are *Standard Deviation* which occur if one assumes a normal distribution for a column in numeric values and the column contains values which fall outside two standard deviations. They implement detection for their smells in a stand-alone tool called SmellSheet

Detective which can analyze spreadsheets from the online office environment Google Docs.

In more recent work Barowy et. Al [21] take a very similar but more formalized and thorough 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. Such values are likely either very important or erroneous. While "Data Debugging" is a valid term, we think the term "Data Smell" is more valid because it shows the similarity to code smells and its similar potential such as locating weak points in the artifact and possibility for refactoring.

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.

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. **TODO: Is this the best place for the disclaimer?**

TODO: Katie asked Chris for a bit of feedback - waiting on response

A. Future Opportunities in the Domains Studied

In a qualitative analysis of the end-user preferences for Yahoo! Pipes programs, the participant responses showed preference toward programs with familiar elements [22]. This opens an opportunity and perhaps a need to define code smells and refactorings based on end-user experience.

B. Future Opportunities in Other EUP Domains

TODO: education (kodu, scratch, alice), mathematics (matlab, mathematica), ...

C. 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]. **TODO: Another lesson from spreadsheets?** 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., [23]). Paired with the end-user evidence, a stronger case can be made to integrate automated smell detection in many domains.

TODO: Other data flow languages could benefit from *normalize order of operations* to improve understandability (as it does with YP).

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

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

[15] 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.

[16] 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.

[17] 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.

[18] 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

[19] 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

[20] 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.

[21] 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.

[22] 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.

[23] 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