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*—**

***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 [4] and 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.**

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], spreadsheets [3], [5], [12], and dataflow programs [6], all of which are end-user programming domains.

III. Smells in end-user artifacts

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

Another 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 or by analyzing publicly available artifacts [4].

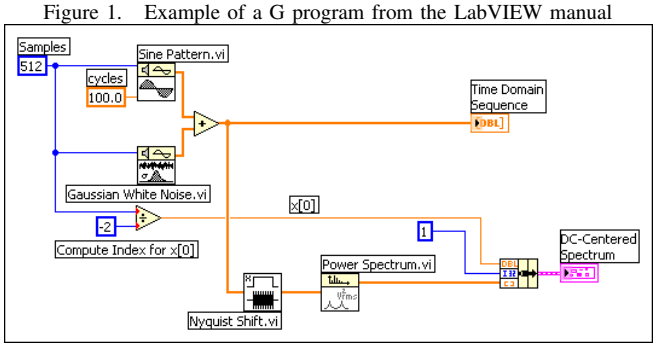
This section provides an overview of different smells that researchers have found to be applicable to end-user artifacts.

A. Smells in 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.

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

Figure 1. Example of a G program from the LabVIEW manual



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. 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 and the problems associated with programs with too much layers or indirection, sometimes referred to as *Lasanga Code*.

Cunha et. al [15] claim to define code smells for spreadsheet, but they take a different approach and consider the data contained in a spreadsheet. However, it is doubtful if these smells qualify as code smells because they are mostly related to user input and not program logic. This also causes them to not have an equivalent in general purpose programming languages. We refer to related work section VI-A.

B. Smells in LabVIEW programs

LabVIEW is a proprietary 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 1. 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.

Some users claim they are two to three times as productive in G as they are in general-purpose alternatives like C or FORTRAN [6]. However, G programs do not always perform well, which has motivated Chambers and Scaffidi [6] to define smells in G and implement heuristics to identify these smells, where they particularly focus on smells that can indicate performance problems.

The thirteen smells identified roughly fall into three categories. The first three are smells specific to the environment, for example the "*Sequence instead of State machine*" smell which is a smell because state machines can be much better optimized by the G compiler.

The second and third categories are more broadly applicable, with the second category of two smells being smells you could find a any program that uses concurrency. One can easily see how "*Non-reentrant VI*" is a smell in a language that heavily relies on concurrency and parallelization since such a VI will require blocking and thus possibly slows down the program, similar to how a non-reentrant function could be a problem in a concurrent program written in a general purpose language.

The third and largest category with eight smells are those that are applicable to general purpose languages. "*Multiple Nested Loops*" is a smell in almost any language that supports loops, and "*Build Array inside Loop*" would be similar to allocating multiple small segments of memory with C's `malloc` inside a loop with potentially equally bad performance implications.

C. Smells in Yahoo pipes

In [4], Stolee and Elbaum describe a number of smells. **TODO: more summary here**

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.

A. Refactoring spreadsheets

The first work on spreadsheet refactoring was done by Badame and Dig [16]. They define six refactorings applicable to spreadsheets, although the corresponding smells are only implied and not explicitly defined and implement these in an Excel plugin called *REFbook*. Three of these have to do with preventing errors, like *Guard Call* which

places an `IF` around an expression which can result in an error such as a division by potentially zero. The other two directly correspond to known smells. *Extract Row or Column* or more generally *Extract Subformula* is a refactoring with which part of a formula is extracted to a new cell. This solves the *Multiple operations* smell, which is analogous to the *Long Method* [13] smell. A similar operation is the *Extract Literal* refactoring, in which a literal value is extracted formulas into a different cell. This can solve the *Magic Number* smell. The last one, *Replace Akward Formula* is very specific and transforms multiple additions into a single `SUM` operation.

Hermans and Dig [12] take a more generic approach and provide a way to transform spreadsheet formulas to other spreadsheet formulas, in a way that is very similar to how a regular expression or patterns works in some modern programming languages. They implement this in an Excel plugin called BumbleBee. To define a transformation rule one defines a formula to match and a formula to replace it with. While every Excel 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 *Guard Call* [16] refactoring by transforming a formula into one which prevents division by zero. In this example F can capture (bind to) any valid Excel Formula. With this approach it is possible to create a rule for all refactorings defined by Badame and Dig [16], but also all refactorings as long as they do not require more intricate capturing groups.

B. Refactoring LabVIEW programs

Chambers and Scaffidi [6] do not provide support for automatic refactorings in their tool, instead relying on the users to know how to fix the indicated smell themselves. However, the LabVIEW programming language G is a visual dataflow language. Sui et. al [17] 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.

C. Refactoring Yahoo Pipes

V. Empirical evidence

A. LabVIEW

Chambers and Scaffidie implemented their smell-finding heuristics in a tool integrated into LabVIEW and evaluated it with 26 experienced LabVIEW programmers divided in two groups of thirteen of which only one had access to their tool.

The results were promising, with all thirteen tool-users being able to identify the problems in the given programs and twelve being able to fix the problem. In contrast only five out of thirteen non-tool-users were able to identify the problem and three out of five being able to fix the problem, although the latter is statistically insignificant. Tool users also needed 34% less time to identify the problem and 65% less time to fix the problem, but these results were not statistically significant. This evaluation shows that experienced end user programmers can benefit from tool support in identifying problems.

Table 1: Impacts of Smells on End-User Programmers

Experiment 1	75 observations		
Which one of these two pipes would you prefer?	**Smelly Pipe**	**Clean Pipe**	**Same**
	24%	63%	13%
Experiment 2	16 observations		
What is the output of this pipe?	**Correct Answer**		
	Smelly Pipe	**Clean Pipe**	
	67%	80%	

Figure 2. End-users showing a preference for non-smelly pipes from [4]

B. Pipes

In addition to the smells, they present evidence that users of Yahoo pipes prefer non-smelly pipes over smelly ones, as shown in Figure 2.

VI. 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 [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. 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 [18] 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.

B. Testing end user-programs

TODO: expand

Refactoring is linked to testing because refactoring an artifact is made safer and easier by the existence of a good test suite in said artifact.

TODO: Mention expector

VII. Discussion

VIII. 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] 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.

[15] 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.

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

[17] 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] 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.