Refactoring meets Spreadsheet Formulas

Sandro Badame, Danny Dig
University of Illinois
{badame1,dig}@illinois.edu

Abstract—The number of end-users who write spreadsheet programs is at least an order of magnitude larger than the number of trained programmers who write professional software. We studied a corpus of 3691 spreadsheets and found that their formulas are riddled with the same smells that plague professional software: hardcoded constants, duplicated expressions, unnecessary complexity, and unsanitized input. These make spreadsheets difficult to read and expensive to maintain. Like automated refactoring in the object-oriented domain, spreadsheet refactoring can be transformative.

In this paper we present seven refactorings for spreadsheet formulas implemented in RefBook, a plugin for Microsoft Excel. To evaluate the usefulness of RefBook, we employed three kinds of empirical methods. First, we conducted a User Survey with 28 Excel users to find out whether they preferred the refactored formulas. Second, we conducted a Controlled Experiment with the same 28 participants to measure their productivity when doing manual refactorings. Third, we performed a Retrospective Case Study on the EUSES Spreadsheet Corpus with 3691 spreadsheets to determine how often we could apply the refactorings supported by RefBook. The results show: (i) users prefer the improved quality of refactored formulas, (ii) RefBook is faster and more reliable than manual refactoring, and (iii) the refactorings are widely applicable. On average RefBook is able to apply the refactorings in less than half the time that users performed the refactorings manually. 92.54% of users introduced errors or new smells into the spreadsheet or were unable to complete the task.

I. INTRODUCTION

The number of end-users who write spreadsheet programs is at least an order of magnitude larger than the number of trained programmers who write professional software [1]–[3]. Therefore the majority of programming is actually performed by users who do not consider themselves programmers. These spreadsheet users are referred to as end-users [4]. While these end-users are responsible for the maintenance and correctness of their spreadsheets, they have not been trained to develop software and often are not trained in the best practices for spreadsheet maintenance.

We have analyzed 3691 spreadsheets from the EUSES Spreadsheet Corpus [5] to determine the internal quality of the spreadsheets. We found that many formulas have smells, similar to those commonly found in professionally developed software. Some smells degrade the performance, others decrease readability, and others make it harder to change the table in the future. For example: 61% of formulas contain numerical constants that can be extracted. By consolidating all of the constant references in a sheet to a single place, the formula’s readability and maintainability is increased. 13.66% of text columns are good candidates for conversion to a dropdown menu which reduces the possibility of typos occurring in a column and convey to a maintainer the acceptable values for a text column. 61% of formulas can be given descriptive names instead of using anonymous cell references, thus making it easier to understand formulas.

Although smelly formulas may correctly perform their tasks, they are difficult to maintain and can mask errors. Such errors have cost millions of dollars [4]. Researchers [6]–[18] have made continuous strides into finding and displaying errors and smells in spreadsheets. However, there is no work on the removal of smells from spreadsheet formulas.

In professional programing the removal of smells while preserving program behavior is called refactoring [19]. Refactoring is an important part of professional software development. Refactoring has revolutionized how programmers design software: it has enabled programmers to continuously explore the design space of large codebases, while preserving the existing behavior. Software has been found to have an inverse relationship between the number of applied refactorings and software defects [20]. Modern IDEs such as Eclipse, NetBeans, IntelliJ IDEA, or Visual Studio incorporate refactoring in their top-level menu and often compete on the basis of refactoring support.

Professional programmers have the support of refactoring tools. End-users, who are not even trained to maintain software, do not have any refactoring support. We propose to remove spreadsheet smells through the use of automated refactoring, analogous to the practice of object-oriented code.

There is a large number of refactorings for spreadsheet formulas that we could have implemented. However, we want to automate those that are frequently performed but cause frustration, and those that are infrequent but are difficult. We asked these questions to different end users.

We contacted the 750 members of the European Spreadsheet Risks Interest Group [21] that subscribe to the mailing list of professional spreadsheet users, and we exchanged several emails with their Chair, Patrick O’Beirne, the author of an influential paper [14] about best practices for spreadsheets. We also posted on two large forums that have been used by hundreds of thousands of users: the OpenOffice Calc forum [22] that has more than 200 active users online at any time, and the Excel forum [23] that has on average 4,000 active members online at any time. We also asked on the staff mailing list at the CS department at UIUC.
Based on our input, we have implemented RefBook, a plugin for Microsoft Excel, that implements seven refactorings that reliably remove spreadsheet smells.

RefBook implements the following refactorings: Extract Column, Make Cell Constant, Guard Call, Replace Awkward Formula, String To Dropdown, Introduce Cell Name, and Extract Literal. Each refactoring is specialized to remove a particular smell from a spreadsheet. These refactorings increase programmer productivity by performing the refactorings quickly and correctly. Extract Column breaks formulas into smaller components and can reduce the amount of code duplication that exists in spreadsheets. Make Cell Constant makes formulas less error prone and more readable by rewriting the formula to contain $'s that signify that a particular cell or column is constant throughout a set of formulas. Guard Call rewrites a cell formula to have user defined behavior when an error condition occurs. Replace Awkward Formula re-writes formulas using Excel’s built-in functions (e.g., SUM) so that spreadsheets become more uniform and easier to understand. String To Dropdown limits the number of allowed values for a cell to reduce the chance of a typo. Introduce Cell Name removes anonymous cells and replaces them with named cells. Extract Literal removes “magic numbers” from formulas.

To evaluate the usefulness of RefBook, we employed three kinds of empirical evaluation methods. First, we conducted a User Survey with 28 Excel users to find out whether they preferred the refactored formulas. Second, we conducted a Controlled Experiment with the same 28 participants to measure their productivity when doing manual refactorings. Third, we performed a Retrospective Case Study on the EUSES Spreadsheet Corpus with 3691 spreadsheets.

This paper makes the following contributions:

1) To the best of our knowledge, we are the first to present refactorings in the domain of spreadsheet formulas.

2) We present the first refactoring tool for spreadsheet formulas, RefBook, implemented as a plugin for Excel. RefBook currently supports seven refactorings. A demo can be seen at: http://www.youtube.com/watch?v=wGlU6Muvd8I

3) Our three-way evaluation reveals:
   (i) For four out of the seven refactorings users preferred the refactored output. Thus the refactorings improve spreadsheet quality. (ii) On average RefBook is able to apply the refactorings in less than half the time that users performed the refactorings manually. Thus RefBook improves programmer productivity. (iii) 92.54% of users asked to perform the same refactorings introduced errors into the spreadsheet or were unable to complete the task. RefBook makes it easier to apply the refactorings correctly, thus it is more reliable. (iv) The refactorings can be applied to many of the formulas contained in spreadsheets. Thus RefBook is applicable.

II. MOTIVATING EXAMPLE

To illustrate the kinds of refactorings applicable to spreadsheets, we will use the table shown in Figure 1(A). This table tabulates data from a warehouse where four salespersons purchased fruits and resold them for a profit. Each row tabulates the data for one salesperson. Under the table we show the kind of each column and the formulas they compute. Notice that we only show the formulas as they would appear in row 5 (thus referring to cells from row 5), but the formulas for the other rows refer to their respective cells.

The table contains twelve columns: six columns contain literals, and six columns contain formulas that compute on the other columns. Now we briefly explain each column.

Column A is a text column with the names of the sellers. Columns B-E are numerical columns that hold the number of each type of fruit that each salesperson had purchased. Column F is a formula column that computes the price that each seller paid for their fruits. It performs two calculations: sum the number of fruits purchased by each person, then multiply it by $0.50, the purchase price per fruit. Column G is a numerical column that holds the amount of money that each seller collected from selling their fruits. Column H is a formula column that computes the number of fruits sold: it divides the Sold Price by $1, the resale price per fruit. Column I is a formula column that computes the remaining...
fruits that each salesperson still has. This column performs two calculations: sum the total number of fruits purchased by that person, then subtract their number of fruits sold. Column J is a formula column that computes Income as the difference between Sold Price and Total Price. Column K is a formula column that computes the return on investment from Income and Total Price. Column L is a text column containing the favorite fruit as reported by the salesperson. This column should only contain the names of real fruits, not arbitrary text.

There are several “smells” in this table. Some smells degrade the performance, others decrease readability, and others make it harder to change the table in the future.

Take for example the expression B5+C5+D5+E5 which computes the total number of sold fruits. First, this can become more readable if it was replaced with the built-in SUM function (i.e., SUM(B5:E5)). Second, this expression is calculated twice, once in the Total Price column and then again in the Remaining Fruits column. Given that Excel does not cache expression results, but instead does cache cell results, this wastes CPU cycles. Third, since this expression is duplicated between the Total Price column and the Remaining Fruits column, a future change request like introducing a new kind of fruit and its afferent column, requires changing the expression in the two columns. Like in the case of professional programming, duplication increases the maintenance effort.

Also notice that the table contains two constants, 0.5 and 1. First, constants make the formulas unreadable: another co-worker who inspects the table will have to guess what is the meaning of these constants (i.e., purchase price and resale price). Second, constants make it tedious to perform maintenance tasks: if we wanted to change the purchase and resale price, we will have to manually find and update 8 cells. Performing a find-and-replace for the constant throughout the entire column of formulas.

Also notice that the Favorite column can contain any arbitrary text, even the ones that are not fruit names, for example by a typo (e.g., “Appls”). This affects the readability of the column for humans. This is even worse for other automated tools: macros or other programs that read such erroneous values won’t work.

Also notice that all formulas that refer to static cells use the “fixed column” format. For example, the formula in H5 refers to the static cell G5. Adding the $ can make cell references more resistant to errors when the spreadsheet is modified, e.g., when the formula is dragged down a column.

Dragging a formula in excel copies the formula into the adjacent cell and changes the cell references in the formula to reference the new row or column that the cell was dragged into. For example, dragging A1 down one row will insert the formula A2 into the new row. While the ability to drag a formula is very useful in practice, it leads to smelly and possibly erroneous spreadsheets when not used carefully.

Bugs can occur when dragging a formula that contains a reference to a cell that is constant throughout the entire table. Dragging this formula will also increment the reference to the constant cell. This means that the dragged formulas will refer to a different cell than the one intended by the user.

The $ signs also serve as a form of documentation about the formula: the $ signs highlight which cell references are constant throughout the entire column of formulas.

Figure 1(B) shows the same table after we applied several refactorings. We added the Total Fruits column to hold and intermediate calculation of total number of purchased fruits. The Total Price and Remaining Fruits formulas are modified to reference the Total Fruits column.

We moved the 0.5 and 1 constants from Total Price and Fruits Sold columns into their own cell and named them PurchasePPF and SoldPPF respectively.

We changed the ROI column to sanitize its input against a division by 0. If the number of sold fruits was 0 then the ROI would display a “divide by zero” error. With the guarded formula, the “Unknown” text is displayed. The advantage is that it notifies the reader that the formula was written with the error case in mind, and that the error case is handled.

We changed the Favorite column to use a dropdown menu for valid values instead of arbitrary text. Using a dropdown also makes the user aware that there are a fixed number of values that are allowed in the column.

Across all of the formula cells, we updated the cell references to use the $ on their column identifier.

**Figure 1:** Architecture of {\textit{RefBook}}.

---

**III. HIGH LEVEL OVERVIEW OF RefBOOK**

**A. Typical use of RefBOOK**

Users interact with {\textit{RefBOOK}} from within Excel. We describe a typical use of {\textit{RefBOOK}} using our motivating example from Figure 1: replace the formulas in the Total Price column with formulas that contain constant cell references. To perform this refactoring the user selects cells F2-F5 from the Total Price column, right-clicks on the selection to bring up the context menu (which shows {\textit{RefBOOK}} on the top), then selects the Make Cell Constant option in the menu. {\textit{RefBOOK}} then replaces the highlighted formulas with formulas that contain the $ sign.

**B. Life-cycle of a Refactoring**

{\textit{RefBOOK}}’s architecture consists of three major components: Excel plugin, Ludwig [24], and the refactoring engine. The
Excel plugin is the front-end. Users only interact with the Excel plugin component of RefBook. The Excel plugin creates a separate process for the back-end, Ludwig and the refactoring engine. The back-end calculates the corresponding changes for each refactoring, and sends them to the Excel plugin that applies them on the spreadsheet.

Ludwig [24] is an off-the-shelf component that given a grammar for a language (e.g., the grammar for Excel formulas – see [25]) generates a Java library for parsing that grammar. The Abstract Syntax Tree (AST) generated by Ludwig supports manipulation of the AST while preserving the formatting for the remainder of the formula. The refactoring engine is a Java application that takes as input the name of the refactoring to perform, the table that is the target of the refactoring, and the user input. It performs the AST transformations, e.g., adding, removing, updating cells. Then it outputs commands for the Excel plugin to perform on the Excel spreadsheet. The output of the refactoring engine is an ordered list comprised of commands of this kind:

- INSERTCOLumn columnIndex
- INSERTROW rowIndex
- SETCELL columnIndex, rowIndex, content
- NAMECELL columnIndex, rowIndex, name

The major advantage of this 3-tier architecture is that RefBook is extensible to other spreadsheet tools, beyond Microsoft Excel. If we were to implement refactoring support for OpenOffice Calc [26] then we would have to reimplement two AST nodes, and it returns a boolean value. Notice that if the nodes contain other child AST nodes, the procedure further checks the children for consistency.

### Procedure 1 isCONSISTENT

```
function isCONSISTENT(node1, node2)
    if node1.type == node2.type then
        return False
    end if
    if node1.type == Function then
        if node1.functionName == node2.functionName then
            return False
        end if
        if node1.children.count == node2.children.count then
            end if
            if for i = 0..(node1.children.count - 1) do
                if !isConsistent(node1.children[i], node2.children[i])
                then
                    return False
                end if
            end for
            return True
        end if
    end if
end function
```

Now we describe each of the seven refactorings supported by RefBook. For each refactoring we present an example of the transformation, then we describe in plain text what the refactoring does, then we provide pseudo-code for the algorithm. Here we show the input, the preconditions, and the transformation. We use the same style of behavior-preservation introduced by Opdyke [27], namely we guarantee that the refactored spreadsheet computes the same values as the original spreadsheet when the preconditions are true.

We define one more notation used in the pseudocode of the refactorings: formula.cellReferences denotes the collection of CellReference nodes that the formula’s AST contains.

### C. EXTRACT COLUMN

**Example:** In the motivating example (Fig. 1), we apply the refactoring to the Total Price column. It extracts the expression: `(B5+C5+D5+E5)` from Total Price and Remaining Fruits into a new cell, F5. The new column will contain formulas like `B5+C5+D5+E5`. The Total Price column will hold `F5=.5` and the Remaining Fruits will hold `F5=H5`.
**D. MAKE CELL CONSTANT**

Example: In the motivating example in Fig 1, we apply the refactoring to all of the formula columns: Total Price, Fruits Sold, and Remaining Fruits. RefBook converts the column formulas from (B5+C5+D5+E5) * 0.5, G5/1 and (B5+C5+D5+E5) - H5 to ($B5+$C5+$D5+$E5) * 0.5, $G5/1, and ($B5+$C5+$D5+$E5) - $H5 respectively.

**Description:** The user selects a whole column. RefBook first determines whether any of the cell references can be made constant. It uses the shape of the first formula as the model for all the other formulas in the selection. Then it compares corresponding cell references between pairs of selected formulas and it determines which cell references do not change (the references with $ prefixes). Due to space limitations, we present the pseudocode in [25].

**E. GUARD CALL**

Example: In the motivating example we apply the refactoring to the ROI column. The user supplied the error expression "Unknown". Guard Call converted J5/F5 to IF (G5<>R5, G5/"Unknown").

**Description:** The user selects a formula cell and also provides an expression to be supplied as the error message. RefBook searches for a division operator, and replaces it with a conditional IF, where the condition checks whether the denominator is different than zero, the then branch performs the division, and the else branch displays the error message.

**Procedure 3 GUARD CALL**

**Input:** formula, errMsg

**Preconditions:** $\exists \forall f \in formula \&\& \text{errMsg.isValidFormula}

**Function GUARDCALL**

```plaintext
function GUARDCALL
    binaryOps = formula.collectAll(AnotherExpression)
    for all n ∈ binaryOps\n        Operator = "f"
        guard = "IF(\"on\" + n.Expression + "<>\"0," + n.Parent + "+errMsg) + "n.parent.ExpressionPrimitive = guard
    end for
end function
```

**F. REPLACE AWKWARD FORMULA**

Example: In the motivating example, we apply the refactoring to the Total Price and Remaining Fruits fruits column converting them from (B5+C5+D5+E5) * 0.5 and (B5+C5+D5+E5) - H5 to SUM(B5:E5) * 0.5 and SUM(B5:E5) - H5 respectively.

**Description:** The user selects a formula and RefBook first searches for expressions containing the + or * operator, and at least four operands of consecutive cells. RefBook replaces such long chains with a single SUM(<range>) or PRODUCT(<range>) function.

**G. STRING TO DROPDOWN**

Example: In our motivating example, we apply this refactoring to the Favorite column. The set of valid entries consists of: Apples, Oranges, and Pears.
Procedure 4 REPLACE Awkward Formula
Input: formula
Preconditions: \( \exists x \) s.t. \( x \neq \text{“or”*} \) in formula
\( \exists (\text{cellRef}) \in \text{formula}(\text{cellRef}).\text{cardinality} > 3 \)
function REPLACEAWKWARD
original = formula
refactored = attempt(formula)
while original \neq refactored do
original = refactored
refactored = attempt(original)
end while
end function

awkwardAST = getAwkwardASTNode(expr)
if awkwardAST \neq \text{NONE} then
expr.replace(awkwardAST, fixedAST)
end if
return expr

Description: The user selects a textual column. \textsc{RefBook} finds all the unique text entries in the column. \textsc{RefBook} attaches dropdown menus to each cell in the column.

Procedure 5 STRING TO DROPDOWN
Input: column
Preconditions: \( \forall \text{cell} \in \text{column} \rightarrow \text{cell.isTextual} \)
function STRING TO DROPDOWN
choices = \{ \text{cell} \in \text{column} | \text{x} \notin \text{choices} \}
for all \text{cell} \in \text{column}
do
\text{cell.DropdownOptions} = \text{choices}
end for
end function

STRING TO Dropdown assumes that the user-selected column does not contain any errors. Specifically if typos exist, they also populate the values in the dropdown menu.

H. INTRODUCE CELL NAME

Example: In our second table from the motivating example, we applied this refactoring to the cell G2, and we named it PurchasePPF. The formula in G5, F5+PurchasePPF, uses the new name.

Excel’s “Search and Replace” is not a sound method to use to replace all anonymous cell references in formulas with the named reference. For example, if the text \( \lambda_1 \) was in a text literal then it would be replaced. If \( \lambda_2 \) was referenced as \( $A51 \) “Search and Replace” misses this reference. Without \textsc{RefBook} the end-user programmer would be forced to inspect each cell in the table for correctness. \textsc{RefBook} reliably and correctly finds all references to the cell.

Description: The user selects a cell, and provides a name. \textsc{RefBook} checks whether the new name is not in use, and defines the name. \textsc{RefBook} searches in the entire table for references to the anonymous selected cell, and updates them to the named cell.

I. EXTRACT LITERAL

Example: In the motivating example, we apply the refactoring to the Total Price and Fruits Sold columns. Each of these columns have a “magic number” (0.5 and 1).

Description: User selects a formula. She also selects the actual literal value from that formula to be extracted and provides a name for the cell. \textsc{RefBook} checks whether the new name is not in use. Then \textsc{RefBook} finds an empty cell that it moves the selected literal into, names the new cell. \textsc{RefBook} determines which AST node was selected to be extracted. Then \textsc{RefBook} searches all cells in the table for formulas that are consistent with the user-selected cell, then in all such consistent cell formulas, the AST node that was selected is replaced with a reference to the named cell. Procedure 7 uses the function getPathTo to find the path from the root node of the formula’s AST to the Primitive that contains the literal node. getPathTo helps avoid the case of a literal in a different context being replaced.

Procedure 6 INTRODUCE CELL NAME
Input: anonCell, name, sheet
Preconditions: name.isValid
function INTRODUCE NAME
anonLoc = location(anonCell)
sheet.defineName(anonCell, name)
for all \text{cell} \in \text{sheet}
do
if location(cellRef) = anonLoc then
\text{cell}.Formula.replace(cellRef, name)
end if
end for
end function

function LOCATION(cell)
return (cell.Workbook, cell.Sheet, cell.Column, cell.Row)
end function

Procedure 7 EXTRACT LITERAL
Input: literal, cellName, cell
Preconditions: cellName.isValid
function EXTRACTLITERAL
\text{sheet} = \text{cell.sheet}
\text{path} = getPathTo(cell.formula, literal)
\text{sheet}.defineName(sheet.getUnUsedCell(), cellName)
\text{sheet}[cellName] = literal
for all \text{c} \in \text{sheet.isConstant(c.formula, c.formula)}
do
\text{node} = c.formula.nodeAtPath(path)
\text{c.formula.replace(node, cellReferenceTo(cellName))}
end for
end function

V. EVALUATION

We evaluate the usefulness of the proposed refactorings by answering four research questions:
- Q1: Do the refactorings improve the spreadsheets quality?
- Q2: Can \textsc{RefBook} make the refactoring process more reliable?
- Q3: Do the refactorings improve programmer productivity?
- Q4: Are the refactorings applicable?

All these questions address the higher level question “Is \textsc{RefBook} useful?” from different angles. Quality measures whether the users find the refactored formulas more readable. Reliability ensures that the runtime behavior is not modified and the transformation does not introduce more smells.
Productivity measures whether automation saves human time. Applicability measures how many formulas in real-world spreadsheets can be directly transformed.

A. Methodology

To answer these questions, we employed three different empirical techniques. (i) To assess whether users preferred the refactored formulas, we conducted an online User Survey with 28 Excel users. (ii) To measure refactoring reliability and user productivity when performing manual refactorings, we conducted a Controlled Experiment with the same 28 users. In order to ensure discretion, each participant responded to the survey and performed the change tasks in their own environment. (iii) To determine how often we could apply the refactorings supported by RefBook, we performed a Retrospective Case Study on the EUSES Spreadsheet Corpus [5] with 3691 spreadsheets.

Recruitment: To recruit participants, we advertised to students in the University of Illinois CS105 course. This course is attended by students enrolled in the Business department. In this course students learn how to use Excel for business-related purposes. The participation in the survey was voluntary, and it did not have any relationship with the course evaluation. The successful completion of the survey was rewarded with a $5 Amazon gift card.

Out of the 500 enrolled students, 28 responded to our call. We asked three questions about their experience with Excel. Figure 3 shows the demographics of our participants. Notice that two-thirds of the participants claimed to have more than two years of experience with Excel. All our participants responded within 24 hours from our post.

1) User Survey: Each participant in the User Survey used an Excel document, consisting of 7 sections. Each section focuses on a single particular smell. Each section has two tables: one table that contains the smell and one table where we have previously used RefBook to remove the smell through one of our refactorings.

For each section, we asked two questions about the tables. The first question was a “filter”: we asked a technical question that revealed whether the participant studied the two tables and understood the differences between them. The second question asked which table they would prefer to work with. To eliminate the confounding effect, we randomized the order of appearance between smelly and non-smelly tables.

2) Controlled Experiment: Each participant used an Excel document. The tables contained data about the orchard warehouse, similar with the example shown in our motivating example from Fig. 1. The document contained 7 tasks. Each task has a smelly table, a set of instructions on what to change in the table, a “Start” button, and a “Task Complete” button. Before the “Start” button is pressed, the spreadsheet is read-only. During this time the participant is free to inspect and become familiar with the table, the task that she will perform, and a short, optional tutorial that we designed to present the Excel features she might use.

Once the participant has familiarized with the task and the table, she can press the “Start” button. When a participant presses the “Start” button, the table becomes editable, and the participant can perform the changes. A timer records the time taken to perform the changes. When a participant completes the task, she presses the “Task Complete” button which stops the timer and moves her onto the next task.

After we received their online submissions, we processed each document to record the time it took participants, and whether they performed the tasks correctly. We also applied ourselves RefBook to complete the same tasks, and then we compared our results with the participants’ results.

3) Retrospective Case Study: To determine the applicability of our refactorings we analyzed the EUSES Spreadsheet Corpus’s 3691 spreadsheets to find out how many formulas have smells that can be fixed by our refactorings. We chose the EUSES Spreadsheet Corpus because it is regarded as the most mature, representative corpus of spreadsheets. At least 13 published papers [5] have used it to draw conclusions about spreadsheet programming. This corpus contains 206355 tables and 495578 distinct formulas.

First, we had written a tool to find the tables in all the spreadsheets. In real-world spreadsheets, tables are (i) often surrounded by documentation, (ii) do not begin at the top and left-most cell, and (iii) multiple tables are scattered throughout the spreadsheet. Our tool parses the corpus spreadsheets using the Apache POI [28] Java library. Due to limitations in the Apache POI [28] library, our tool could not parse 234 spreadsheets of the 3925, so we retained 3691. To find the individual tables that exist within a sheet, we used the algorithm described in [16].

Then we implemented a collector tool, customized for our refactorings. The collector calculates how many cells in each table manifest a particular smell that can be fixed by a particular kind of refactoring.

B. Results

Quality: Table I shows for each refactoring kind, how many participants preferred the smelly or the refactored formulas.

For 4 out of the 7 refactorings, the majority of participants preferred the refactored formulas. For one refactoring, String to Dropdown, the majority did not have any preference. For 2 refactorings, the majority preferred the smelly formulas. For the Introduce Cell Name, the participants preferred the table that contained the anonymous cells. Since 82.61% of the participants were able to correctly answer the filter question about the table, we are confident that they understood the table. This corroborates another study [29] where end-users working with Yahoo pipes preferred seeing all the pipes at once instead of abstracting functionality to another pipe.

When judging whether the refactorings improve the readability for end users, we assume that their opinion reflects
Table I: Preferences of users toward refactorings.

<table>
<thead>
<tr>
<th>Refactoring Kind</th>
<th>Refactor</th>
<th>Prefer No</th>
<th>Prefer Resp</th>
<th>Pass Filter</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExtractColumn</td>
<td>17.39%</td>
<td>47.83%</td>
<td>21.74%</td>
<td>13.04%</td>
</tr>
<tr>
<td>MakeCellConstant</td>
<td>21.74%</td>
<td>52.17%</td>
<td>13.04%</td>
<td>13.04%</td>
</tr>
<tr>
<td>StringToDropdown</td>
<td>8.70%</td>
<td>21.74%</td>
<td>56.52%</td>
<td>13.04%</td>
</tr>
<tr>
<td>ReplaceAwkward</td>
<td>4.35%</td>
<td>52.17%</td>
<td>26.09%</td>
<td>17.39%</td>
</tr>
<tr>
<td>GuardCall</td>
<td>47.83%</td>
<td>13.04%</td>
<td>26.09%</td>
<td>13.04%</td>
</tr>
<tr>
<td>IntroduceName</td>
<td>52.17%</td>
<td>4.35%</td>
<td>26.09%</td>
<td>17.39%</td>
</tr>
<tr>
<td>ExtractLiteral</td>
<td>17.39%</td>
<td>60.87%</td>
<td>8.70%</td>
<td>13.04%</td>
</tr>
</tbody>
</table>

Table II: Safety & Productivity of manual vs. automated refactorings.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ExtractColumn</td>
<td>100 (26+74)</td>
<td>36</td>
<td>18</td>
<td>16</td>
</tr>
<tr>
<td>MakeCellConstant</td>
<td>30 (26+4)</td>
<td>37</td>
<td>17</td>
<td>9</td>
</tr>
<tr>
<td>StringToDropdown</td>
<td>82 (82+0)</td>
<td>217</td>
<td>169</td>
<td>9</td>
</tr>
<tr>
<td>ReplaceAwkward</td>
<td>47 (47+0)</td>
<td>68</td>
<td>42</td>
<td>22</td>
</tr>
<tr>
<td>GuardCall</td>
<td>82 (82+0)</td>
<td>67</td>
<td>28</td>
<td>31</td>
</tr>
<tr>
<td>IntroduceName</td>
<td>82 (56+26)</td>
<td>79</td>
<td>50</td>
<td>30</td>
</tr>
<tr>
<td>ExtractLiteral</td>
<td>91 (34+57)</td>
<td>42</td>
<td>23</td>
<td>18</td>
</tr>
</tbody>
</table>

The table shows that performing the refactoring with REFBOOK is faster than performing it manually, the improvements ranging from 2.2x to 24x. This is a conservative lower bound; we expect the productivity difference to be even more dramatic in practice. First, real-world tables contain more rows than the 15 rows in our controlled experiment. Second, the faults committed by the participants were typically errors of omission: many participants had applied the refactoring incompletely. Had they applied the complete refactoring, this would have taken them even more time.

Applicability: Based on the EUSES Spreadsheet Corpus, we present the applicability of individual refactorings.

There are many formulas in a spreadsheet that are dragged down a column or across a row. However, the user only created one single formula, the rest only differ by the cell references. If we took these dragged cells into account, our results will be skewed by the amount of dragged cells that exist in each table. To prevent this, we define and compute metrics only over distinct formulas (defined in Section IV).

ExtractColumn. First, we measured the formula complexity. We introduce the following metric to measure a formula’s complexity: the sum of the number of binary operators and function calls that a formula contains. For example:

\[
\text{IF}(G5 < 0, K5/G5, "Unknown")
\]

has a complexity of 3 (one function call, i.e., \text{IF}, plus two binary operators, i.e., \text{not equals} and division). A formula that contains only a single reference to another cell, number, or text, has a complexity of 0.
We found that 10.1% of distinct formulas have a complexity of 0, 57.71% have a complexity of 1, 18.49% have a complexity of 2, 11.97% have a complexity of 3, 1.73% have a complexity greater than 3. Formulas that have complexity larger than 1 (32.19% of all distinct formulas), are candidates for the \texttt{EXTRACT COLUMN}, which reduces the complexity of a formula by breaking it into smaller sub-formulas.

We also compute the amount of duplication that exists between distinct formulas in a table. We calculate the amount of duplication by counting the number of times an AST node is repeated in a table. 72.89% of the formulas contain no duplication. The remaining 27.11% contain some amount of duplication, thus are candidates for \texttt{EXTRACT COLUMN}.

\textbf{MAKE CELL CONSTANT} We apply this refactoring on every distinct formula and recorded the number of cell references that were successfully made constant. We found that 23.28% of the formulas did not change when we applied the \texttt{MAKE CELL CONSTANT} refactoring. 9.03% of the formulas had a single $ prefix added to a cell reference, 19.35% of the formulas had two places where $ was added to cell references (e.g., $A$5), 3.24% of the formulas had three $ added to cell references, 32.64% of the formulas had four $ added to cell references (e.g., $A$5 + $B$5).

\textbf{GUARD CALL} We found that \texttt{IFERROR}, \texttt{ISBLANK}, and \texttt{ISNUMBER} are among the top 10% most used Excel functions. This shows that explicit error handling is a popular technique.

\textbf{REPLACE AWKWARD FORMULA} We found that 15.73% of all distinct formulas use the \texttt{SUM} function. This shows that end-users understand it and like to use it.

\textbf{STRING TO DROPOWN} We computed the number of duplicated entries that exist in a column of text values. We found that 86.34% of the text columns had no duplication of text values, 6.99% of the text columns repeated up to 50% of the text entries, and 6.67% repeated between 51% and 99% of their text entries. 95% of the text columns with 51% or more cells repeated can be converted into a dropdown menu with 10 or less different choices. We found that 13.66% of text columns are strong candidates for \texttt{STRING TO DROPOWN}.

\textbf{INTRODUCE CELL NAME} We found that 61% of formula columns refer to at least one common anonymous cell that is a good candidate for being named. For example, column \texttt{C} contains formulas of the type: \texttt{A1+B0, A2+B0, A3+B0}; cell \texttt{B0} is a good candidate for \texttt{INTRODUCE CELL NAME}. We also found that in such columns, on average 2.21 cells could be named.

\textbf{EXTRACT LITERAL} Among formula columns, we found that 61% of them referred to the same numerical constants. For example, column \texttt{S} contains formulas of the type: \texttt{C1+12, C2+12, C3+12}; constant 12 can be extracted into a separate cell. Of the columns where \texttt{EXTRACT LITERAL} can be applied, on average 2.09 numerical constants can be extracted. We also found that 0.06% of formula columns referred to the same string literal. From these columns, on average 1.84 string literals can be extracted.

\section{Discussion}

\textbf{Future Extensions:} There are new refactorings that we could implement, some being trivial extensions of the current refactorings. For example, for implementing \texttt{EXTRACT ROW}, we could use the same pseudocode we used for \texttt{EXTRACT COLUMN} in Procedure 2 where we swap the columns with rows and vice-versa. Additionally, we could also implement refactorings that span multiple sheets.

Also, we could make our prototype implementation more robust. For example, it currently does not check whether the user selects a subexpression that transcends the boundary of operator precedence. A user could select $2+3$ from the $6*2+3$ formula and apply \texttt{EXTRACT COLUMN}. An industrial-strength implementation should raise a warning that the new formula will compute value 30 instead of 15.

\textbf{Threats to Validity:} One could raise objections to our empirical findings based on the distribution of participants. 32% of our participants are self-reported novice Excel users. If we had more novices, they might have preferred the smelly version of the formulas, thus influence the outcome of our findings. However, we believe proficient Excel users can provide more valuable feedback about spreadsheets quality, similarly with how proficient Java programmers provide more valuable feedback for the Eclipse refactoring engine.

A very useful validation of refactorings could measure whether users find the refactored formulas easier to maintain. However, this is very hard in a study with students, because they never maintain their formulas.

\section{Related Work}

Erwig [12] proposes to apply techniques and tools from professional programming to end-users. Erwig specifically advocates for better error reporting, debuggers and static type checking [30]. He does not mention applying the refactoring techniques from professional programming.

Guidelines for creating clean, consistent and non-smelly spreadsheets have been proposed by others [13], [14].

PUP [17] and ASAP Utilities [18] are tools that add many features to Excel including some basic formula manipulation. The “Error Condition Wizard” in PUP and “Custom Error Message” in ASAP Utilities are similar in spirit to our \texttt{GUARD CALL} refactoring. The major difference is that these tools do not let users type in arbitrary expressions to be executed in the else branch, but they limit the type to Strings, unlike \texttt{REPLACE}, that allows any arbitrary expression. Also, our \texttt{GUARD CALL} infers the check for erroneous behavior, it does not react to an already existing error. This gives users more flexibility to define the action that should be taken for bad input.

ASAP Utilities includes a "Change formula reference style" operation and Excel has a feature that adds $ sign to cell references that is similar in spirit to our \texttt{MAKE CELL CONSTANT}. However, they cycle through the selected cells and blindly add $ to every single cell reference. Our \texttt{MAKE CELL CONSTANT} is different from the above alternatives because it intelligently
determines which cells should be made constant based on their usage in similar formulas from the user selection.

"What You See is What You Test" (WYSIWYT) [7] is a testing tool that helps end users find bugs in their spreadsheets. WYSIWYT estimates a cell’s correctness based on user input. WYSIWYT does not offer any automation to correct cells that are found to be incorrect.

Cunha et al. [8] detect data in spreadsheets that are outliers from the typical entries. These outliers are referred to as being "smelly". Their work does not apply to finding spreadsheet formula smells. EREF BOOK removes smells from formula cells not from data cells.

Hermans et al. [9]–[11] implemented tools to detect spreadsheet smells and visualize them. Their work is complementary to ours, since it focuses on making spreadsheet smells apparent through visualizations but does not support removing the smells. Also their work focuses on inter-table smells, whereas we focus on intra-table smells and their correction.

Harris et al. [16] implemented a tool that infers spreadsheet transformations by parsing a small example of the transformation and then extrapolating that example to an entire table. Their work focuses on transforming the layout of data cells and does not take cell formulas into account. EREF BOOK has a predefined set of refactoring while their tool infers a new transformation for every example.

The inspiration for our project comes from research on refactoring for end-user programming in the context of Yahoo Pipes [29]. While both Yahoo Pipes refactoring and EREF BOOK target end-users, the environments of these users are different. Spreadsheets have different smells and require a different set of tools to remove these smells.

VIII. CONCLUSIONS

End users working with spreadsheets make the same poor choices that professional developers make and have to pay the same "technical debt" that professional programmers pay during maintenance.

We designed, implemented, and evaluated EREF BOOK, the first refactoring tool for spreadsheet formulas. It currently supports 7 refactorings that eliminate smells in spreadsheets.

Our three-pronged evaluation (case study of the EUSES Spreadsheet Corpus, user survey and controlled experiment with 28 participants) concludes that the refactorings supported by EREF BOOK are widely applicable, increase programmer productivity, increase the reliability of transformations, and increase the quality of spreadsheets. More research is needed to find why end users do not feel comfortable with abstraction and how to create tools that they can embrace.

Acknowledgements: The authors thank Cosmin Radoi, Semih Okur, and the anonymous reviewers for constructive feedback. Jeff Overbey for providing assistance with Ludwig, Patrick O’Beirne for his advice on selecting refactorings to automate, the 28 participants in our study, and Microsoft for partially funding this research through a SEIF award.

REFERENCES