Spreadsheets are a widely used end-user programming tool. Field audits have found that 80-90% of spreadsheets created by end users contain textual and formula errors in spreadsheets. Such errors may have severe negative consequences for users in terms of productivity, credibility, or profits. To solve the problem of spreadsheet errors, researchers have presented manual and automatic error detection. Manual error detection is both tedious and time-consuming, while automatic error detection is limited to only finding some formula error categories such as formula reference errors. Both approaches do not provide the optimum result in error detection.

We have tested a new error detection approach by detecting bad smells in spreadsheets, which is an indication that an error might be present. Originally developed for object-oriented programming, examples include the large class, and the lazy class. We have adapted the concept of bad smells to spreadsheets. Each bad smell detector might indicate an issue in the spreadsheet, but the indication is not definitive, since the user must examine the spreadsheet and make a final judgment about whether an error is actually present. We evaluated 11 bad smell detectors by analyzing the true positives against the false positives. The result shows that six detectors can highlight some error categories, such as categorical errors and typographical errors.
Detecting Bad Smells in Spreadsheets

by
Atipol Asavametha

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented June 15, 2012
Commencement June 2013

APPROVED:

Major Professor, representing Electrical and Computer Engineering

Director of the School of Electrical Engineering and Computer Science

Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Atipol Asavametha, Author
ACKNOWLEDGEMENTS

Firstly, I would like to thank my advisor, Asst. Prof. Christopher Scaffidi, for his superb mentoring and for providing me with thought-provoking comments on the drafts of this thesis. He has given me support and guidance throughout the long and difficult process, and for that, I sincerely thank him for his kindness and patience. If there should be a mistake or an error in the thesis, the flaw is mine and mine alone. Additionally, I would like to express my gratitude to Prof. Bella Bose and Asst. Prof. Carlos Jensen who have given me an opportunity to be a part of the School of Electrical Engineering and Computer Science at Oregon State University. It has been a priceless academic experience.

Other than the faculty members at Oregon State University, I would like to thank Ms. Sudarat Sathapanachai, Asst. Prof. Sumalee Pisitkasem, and Dr. Wasan Pattara-atikom for their recommendation letters. I thank all the fellow students in the School of Electrical Engineering and Computer Science, especially Aniket Dahotre, Thuan Hong Duong Ba, Shalini Shamasunder, and Ali Aburas, for their friendship and their help and support. Furthermore, I thank the Thai students in Oregon, especially the five exchange students from Chulalongkorn University. More importantly, I would like to thank Aim Sinpeng and Prakaiwan Vajrabhaya who helped me edit the final draft of my thesis. Finally, I thank the "Asavametha" family and my future wife, Tarinee Lubpairee, for their encouragement and understanding. No words could ever precisely describe my gratitude for their support.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2 Related work and background</td>
<td>5</td>
</tr>
<tr>
<td>2.1 Extracting rectangular regions</td>
<td>5</td>
</tr>
<tr>
<td>2.2 Finding functional dependencies</td>
<td>7</td>
</tr>
<tr>
<td>2.3 Detecting errors in spreadsheets</td>
<td>9</td>
</tr>
<tr>
<td>2.4 Bad smells</td>
<td>11</td>
</tr>
<tr>
<td>3 Approaches</td>
<td>14</td>
</tr>
<tr>
<td>3.1 Extracting rectangular regions</td>
<td>15</td>
</tr>
<tr>
<td>3.2 Pre-processing step</td>
<td>18</td>
</tr>
<tr>
<td>3.2.1 Row Sets</td>
<td>20</td>
</tr>
<tr>
<td>3.2.2 Extract footer rows</td>
<td>22</td>
</tr>
<tr>
<td>3.2.3 Extract header rows</td>
<td>23</td>
</tr>
<tr>
<td>3.2.4 Extract data rows</td>
<td>28</td>
</tr>
<tr>
<td>3.3 Finding functional dependencies</td>
<td>28</td>
</tr>
<tr>
<td>3.4 Finding keys and the best key</td>
<td>29</td>
</tr>
<tr>
<td>3.5 Detecting bad smells</td>
<td>30</td>
</tr>
<tr>
<td>3.5.1 Bad smell detectors that just count rows and columns</td>
<td>32</td>
</tr>
<tr>
<td>3.5.2 Bad smell detectors that rely on simple formula analysis</td>
<td>34</td>
</tr>
<tr>
<td>3.5.3 Bad smell detectors that use complicated formula analysis</td>
<td>35</td>
</tr>
<tr>
<td>3.5.4 Bad smell detectors for finding hidden or duplicate cells</td>
<td>38</td>
</tr>
</tbody>
</table>
TABLE OF CONTENTS (Continued)

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 Evaluation</td>
<td>40</td>
</tr>
<tr>
<td>4.1 Testing smell detectors</td>
<td>40</td>
</tr>
<tr>
<td>4.2 Results</td>
<td>43</td>
</tr>
<tr>
<td>5 Conclusion and future work</td>
<td>52</td>
</tr>
<tr>
<td>5.1 Conclusion</td>
<td>52</td>
</tr>
<tr>
<td>5.2 Future work</td>
<td>52</td>
</tr>
<tr>
<td>Bibliography</td>
<td>54</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>An example of a rectangular region.</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>An example of a functional dependency $X \rightarrow Y$.</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>Execution times in seconds for correlated data.</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>Descriptions of bad smells in object-oriented environment.</td>
<td>12</td>
</tr>
<tr>
<td>5</td>
<td>Process for detecting bad smells in spreadsheets.</td>
<td>15</td>
</tr>
<tr>
<td>6</td>
<td>Algorithm for extracting the rectangular region.</td>
<td>17</td>
</tr>
<tr>
<td>7</td>
<td>An example of multiple rectangular regions in the spreadsheet.</td>
<td>18</td>
</tr>
<tr>
<td>8</td>
<td>An overview of pre-processing step.</td>
<td>18</td>
</tr>
<tr>
<td>9</td>
<td>An example of a rectangular region.</td>
<td>19</td>
</tr>
<tr>
<td>10</td>
<td>Algorithm for pre-processing step.</td>
<td>21</td>
</tr>
<tr>
<td>11</td>
<td>Algorithm for extracting footer rows.</td>
<td>23</td>
</tr>
<tr>
<td>12</td>
<td>Algorithm for extracting header rows.</td>
<td>24</td>
</tr>
<tr>
<td>13</td>
<td>Algorithm for computing non-empty referred rows.</td>
<td>26</td>
</tr>
<tr>
<td>14</td>
<td>Algorithm for computing empty referred rows.</td>
<td>27</td>
</tr>
<tr>
<td>15</td>
<td>Algorithm for indicating a redundant functional dependency.</td>
<td>29</td>
</tr>
<tr>
<td>16</td>
<td>Bad smells are used to analyze errors in spreadsheets.</td>
<td>32</td>
</tr>
<tr>
<td>17</td>
<td>Algorithm for <em>Data Clumps Detector</em>.</td>
<td>34</td>
</tr>
<tr>
<td>18</td>
<td>Algorithm for <em>Long Member Detector</em>.</td>
<td>34</td>
</tr>
<tr>
<td>19</td>
<td>Algorithm for <em>Long Message Chain Detector</em>.</td>
<td>35</td>
</tr>
<tr>
<td>20</td>
<td>Algorithm for <em>Long Parameter List Detector</em>.</td>
<td>36</td>
</tr>
<tr>
<td>21</td>
<td>Algorithm for <em>Feature Envy Detector</em>.</td>
<td>37</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>22.</td>
<td>Algorithm for <em>Switch Statements Detector</em></td>
<td>37</td>
</tr>
<tr>
<td>23.</td>
<td>Algorithm for <em>Temporary Field Detector</em></td>
<td>38</td>
</tr>
<tr>
<td>24.</td>
<td>Algorithm for <em>Duplicated Cell Detector</em></td>
<td>39</td>
</tr>
<tr>
<td>25.</td>
<td>The summary result of the targeted errors.</td>
<td>42</td>
</tr>
<tr>
<td>26.</td>
<td>The result of <em>Large Class Detector</em></td>
<td>43</td>
</tr>
<tr>
<td>27.</td>
<td>The result of <em>Lazy Class (tuple) Detector</em></td>
<td>44</td>
</tr>
<tr>
<td>28.</td>
<td>The result of <em>Lazy Class (non-key) Detector</em></td>
<td>44</td>
</tr>
<tr>
<td>29.</td>
<td>The result of <em>Data Clumps Detector</em></td>
<td>45</td>
</tr>
<tr>
<td>30.</td>
<td>The result of <em>Long Member Detector</em></td>
<td>45</td>
</tr>
<tr>
<td>31.</td>
<td>The result of <em>Long Message Chain Detector</em></td>
<td>46</td>
</tr>
<tr>
<td>32.</td>
<td>The result of <em>Long Parameter List Detector</em></td>
<td>46</td>
</tr>
<tr>
<td>33.</td>
<td>The result of <em>Feature Envy Detector</em></td>
<td>47</td>
</tr>
<tr>
<td>34.</td>
<td>The result of <em>Switch Statements Detector</em></td>
<td>47</td>
</tr>
<tr>
<td>35.</td>
<td>The result of <em>Temporary Field Detector</em></td>
<td>48</td>
</tr>
<tr>
<td>36.</td>
<td>The result of <em>Duplicated Cell Detector</em></td>
<td>48</td>
</tr>
<tr>
<td>37.</td>
<td>The result of combined multiple detectors</td>
<td>50</td>
</tr>
</tbody>
</table>
Detecting Bad Smells in Spreadsheets

1 Introduction

According to an analysis of statistics from the U.S. Bureau of Labor and Statistics, the number of American workers who use spreadsheets or databases will have increased to more than 55 million by 2012 [4]. End users may include teachers, students, children, accountants, scientists, or anyone wishing to create spreadsheets or databases for their own use [1]. Spreadsheets are commonly used as end-user programming tools to create, for instance, simple programs using conditionals or formulas, to compute data by writing formulas, or to store data during personal tasks.

Many studies of spreadsheets have shown that end users continually create more complex spreadsheets, with formula content and size doubling every three years [5] [7]. Field audits conducted by researchers found that 80-90% of these increasingly complex spreadsheets contain errors [19]. Specifically, 5% of formulas contain errors and redundancy [19]. Such errors can have severe impacts for some users in terms of productivity, credibility, financial information, and loss of income.

Several researchers have sought to address this problem by seeking ways to detect errors manually and automatically. However, there are limitations with existing approaches implemented to detect spreadsheet errors. Manual error detection is both tedious and time-consuming [10] [11]. Semi-automatic error detection methods alleviate this problem (e.g. [3] [10] [15] [21]) but currently have not been extended or tested for use with finding certain kinds of errors such as erroneous duplications of formulas or string values.

Due to such limitations, more versatile automatic error detection approaches for spreadsheets are needed. Specifically, we have chosen to tackle the problem of cell errors by analyzing relationships of cells within rectangular regions. Each rectangular region has to be extracted before analyzing its internal relationships among its columns, and then detecting potential errors in the region. After extracting a rectangular region, an example of which is shown in Figure 1, our approach consists of three steps: extracting functional dependencies among columns, finding the best
key columns, and detecting errors. These steps are summarized below and described in
detail later within this thesis.

<table>
<thead>
<tr>
<th>Employee ID</th>
<th>Project ID</th>
<th>Employee Name</th>
<th>Job Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>103</td>
<td>15</td>
<td>June E.</td>
<td>Programmer</td>
</tr>
<tr>
<td>101</td>
<td>22</td>
<td>John G.</td>
<td>Database Designer</td>
</tr>
<tr>
<td>105</td>
<td>15</td>
<td>Alice K.</td>
<td>Database Designer</td>
</tr>
<tr>
<td>106</td>
<td>25</td>
<td>William M.</td>
<td>Programmer</td>
</tr>
</tbody>
</table>

Figure 1. An example of a rectangular region [14]

First, we extract a rectangular region and find relationships between columns in
the rectangular region. Since the structure of rectangular regions in spreadsheets often
resembles a table in relational database, and since the concept of functional
dependencies can be used to characterize implicit relationships among columns of
relational tables, an existing data mining algorithm was applied for finding functional
dependencies in the rectangular region [18]. We also improved this algorithm to return
the minimum (most concise) functional dependencies in each rectangular region. For
example, our algorithm extracts two functional dependencies, which are \{Employee
ID \rightarrow Project ID, Employee Name, Job Class\} and \{Employee Name \rightarrow Employee ID,
Project ID, Job Class\}, from the example in Figure 1. Roughly translated to natural
language, the first of these functional dependencies can be understood as, “Employee
ID implicitly determines the other columns—if two rows have the same employee ID,
then the other columns also are the same.”

Second, we find the best key of each rectangular region. A key is a set of columns
that implicitly determines all of the other columns; unlike in the examples above, the
key might consist of more than one column. Functional dependencies and heuristics
are used to select one or more columns as a key for the rectangular region.
Researchers have speculated that people often use the left-most column as the key column, perhaps because most languages are left-to-right [3]. Inspired by this, we have enhanced existing key-selection algorithms to use information about each candidate key’s size, its columns’ distance from the leftmost column, and its columns’ contents (including the type and length of cells in the columns). For instance, columns with extremely long strings would typically not be chosen by our heuristic as the key of the rectangular region because we speculate that such strings are difficult to type and therefore probably not used by people as keys when a short number could instead be used. For example, our algorithm chooses Employee ID in Figure 1 as the best key instead of Employee Name, both because Employee ID is further to the left and because it is a short number instead of longer text. Both functional dependencies and the selected key are used to detect errors in the final step.

Finally, our approach identifies areas in the spreadsheet where errors appear likely. Our method for this is inspired by the fact that the structure of rectangular regions is similar to object-oriented programming structure. For example, rows are similar to class instances and columns are similar to class members. The rectangular structure might reveal errors which are analogous to problems in object-oriented programming. In object-oriented programming, there is a term called “bad smells” which is used to track down potential errors [13]. The bad smell is an indication in programming that possibly indicates an area of code is probably hard to maintain and, therefore, could potentially contain a latent error or other problem. A bad smell might indicate something is wrong, but it is not a certainty. Programmers have to look deeper to see if there is a real problem in the source code. There are different types of properties that can be applied for detecting bad smells in object-oriented programs. Each bad smell can be detected by different static analyses. Due to the similarity of the object-oriented programming and rectangular region structure, we expect that detecting similar bad smells will reveal some errors in spreadsheets. Therefore, we have implemented 11 static analyses in the spreadsheet environment by adapting the bad smells from object-oriented programming for which we could identify some
potential candidate analogy to spreadsheets; these included, for example, *Long Parameter List*, and *Long Message Chain* [13]. Some bad smell analyses may be useful for detecting some types of errors, while other analyses may be useless.

To evaluate these static analyses for detecting bad smells in spreadsheets, we tested our approach on spreadsheets from the existing EUSES Spreadsheet Corpus [14]. To summarize the result of our evaluation, we computed true positive and false position rates. True positive shows the fraction of invalid rectangles correctly predicted as errors. False positive shows the fraction of valid rectangles erroneously predicted as errors. We have found that 7 detectors can detect errors in spreadsheets fairly well and might be worthy of further investigation in future work. For example, the *Lazy Class* detector worked particularly well for finding both textual and formula errors. Other detectors were able to find various kinds of textual errors, duplicate cell values, and other specific kinds of errors.

The rest of this thesis is organized in four parts. Section 2 discusses related work that ours complements and/or extends. Section 3 presents our new contributions, including enhancements to existing algorithms and the static analyses behind each of our bad smell detectors. Section 4 shows the results of testing these bad smell detectors on existing spreadsheets and highlights a few that worked especially well compared to the others. Section 5 talks about the conclusions and future research.
2 Related Work And Background

It is common knowledge that spreadsheets contain many errors. Researchers have found that indeed over 80% of spreadsheets have errors [19]. Errors may appear in formula and string cells [3] [19]. Furthermore, spreadsheets may contain duplicate cells or redundant information. These errors should not be taken lightly and have been proven disastrous for some users; for instance, one Texan oil company lost millions of dollars through errors in spreadsheet formulas [19].

The problem of error-prone spreadsheets has been addressed by many research studies (e.g., [3] [15] [19] [21]). As we describe below, our approach builds upon existing algorithms for:

a) Extracting rectangular regions

b) Finding functional dependencies

c) Detecting errors in spreadsheets

d) Bad smells

2.1 Extracting rectangular regions

Researchers have presented approaches for extracting regions in spreadsheets. These approaches are useful because after a rectangular region is identified, then it can be analyzed in the next step to find potential errors. For example, the testing methodology “What You See Is What You Test” (WYSIWYT) infers regions in two steps [15]. The first step determines the similarity of formula cells. The second step involves grouping similar cells into regions. This approach implemented three region inference algorithms for identifying D-Regions, C-Regions, and R-Regions, corresponding to discontinuous, contiguous, and rectangular regions respectively. These regions are identified based on formula similarity in cells.
Of these three algorithms, the one most similar to our own is the R-Regions algorithm, which can find rectangular regions in spreadsheets [15]. This algorithm iterates through all formula cells where each formula cell is merged into the above or below cell depending on the formula similarity. These regions are compared with their neighboring regions. That is, if the neighboring regions have a similar height and formula, these regions will be merged together. However, it is important to note that only similar formula cells in data rows are grouped in the same region; these rectangular regions do not include header cells and textual cells. In fact, researchers found that 40% of cells in spreadsheets contain textual cells [14]. Spreadsheets may have errors in textual cells, but the R-Regions algorithm does not include textual cells in the discontinuous region.

Our approach requires a rectangular table because all cells will be used to analyze relationships among the columns. Spatial analysis algorithms have been used to find errors based on similar relationships; of these, the most similar algorithm to our own is UCheck, which can identify tables in a spreadsheet and detect unit errors [20]. These tables are similar to our rectangular regions because neighboring cells are grouped together in the connected cell area. Implicit unit relationships among columns, based on their column headers, are used to check formulas. As with WYSIWYT, however, UCheck cannot find errors in string data.

Similar to the algorithms above, our approach will identify rectangular regions by finding contiguous cells, but our algorithm will include both textual and formula cells. As with UCheck, our rectangular regions consist of header, data, and footer rows if these rows are adjacent rows. Unlike UCheck, our columns will not be analyzed for unit errors. Instead, cells in the header rows are used to find functional dependencies between columns in each spreadsheet, in the hopes of uncovering new classes of spreadsheet errors.
2.2 Finding functional dependencies

Our approach for finding errors relies on functional dependencies, which are a concept that originated in the context of relational databases [22]. The structure of a relation is identical to a table, which consists of a horizontal header. One column in a table may implicitly determine other columns. This relationship is called a functional dependency, which is an implicit integrity constraint in the relational database. The functional dependency shows the relationship between two sets of attributes in the table. For example, $X \rightarrow Y$ shows that each value of column $X$ can determine the values of column $Y$ (Figure 2). It also denotes that one set $Y$ of attributes is dependent on another set $X$ of attributes: when any two rows of the table that have the same values of $X$, they also have the same values of $Y$. Our approach will use functional dependencies to find candidate keys for each rectangular region. A selected key, along with functional dependencies, will be analyzed for detecting certain potential errors in spreadsheets.

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>X1</td>
<td>Y3</td>
</tr>
<tr>
<td>X2</td>
<td>Y1</td>
</tr>
<tr>
<td>X1</td>
<td>Y3</td>
</tr>
<tr>
<td>X3</td>
<td>Y2</td>
</tr>
</tbody>
</table>

Figure 2. An example of a functional dependency $X \rightarrow Y$

Researchers have presented several techniques for finding functional dependencies: given a rectangular region, these algorithms output a list of columns whose values can be determined by the values in other columns. These algorithms include FUN and TANE [18] [24]. Figure 3 below compares these algorithms (from [18]). $|\tau|$ is the cardinality of the relation (number of rows), $c$ is the rate of data
correlation (number of functional dependencies within the table), and $|R|$ is the number of attributes (number of columns). Empty cells in the table show that experiments were interrupted because execution time exceeded three hours. When the number of attributes or the correlation rate is increased, the gap between execution times of TANE and FUN was increased as well. In addition, when the relation is enlarged, FUN remains efficient and shows better performance than TANE algorithm.

| $c\backslash|R|$ | 10  | 20  | 30  | 40  | 50  | 60  |
|--------------|-----|-----|-----|-----|-----|-----|
| 30%          | FUN | 2.563| 11.035| 25.606| 46.156| 77.960| 131.599 |
|              | TANE| 2.603| 11.316| 27.900| 67.547| 178.616| 505.486 |
| 50%          | FUN | 4.736| 20.739| 50.903| 60.050| 246.594| 584.961 |
|              | TANE| 4.816| 23.964| 132.921| 802.545| 3654.332 |
| 70%          | FUN | 6.879| 30.553| 73.966| 205.357| 404.701| 1038.773 |
|              | TANE| 7.080| 42.050| 269.737| 2069.781 |

Figure 3. Execution times in seconds for correlated data ($|r| = 100,000$) (from [18])

Because of these performance data, we chose FUN as the starting point for our work aimed at extracting functional dependencies from rectangular regions. The FUN algorithm applies a level-wise dynamic programming approach to “grow” increasingly large functional dependencies by using the output of previous iterations (see [18] for details). From the set of functional dependencies, it is straightforward to compute sets of columns that together determine all of the columns [22]. Each such set could serve as a key. For example, in Figure 1, the set {Employee ID} and the set {Employee Name} each would be acceptable keys.

One limitation is that the FUN algorithm might not provide the minimal (most succinct) functional dependencies, nor does it indicate which key set to choose as the primary key. We address these limitations as described in Section 3.
2.3 Detecting errors in spreadsheets

Spreadsheets are used by people to perform a lot of tasks (e.g., financial tasks, reporting tasks, inventory tasks, etc.). Due to their flexibility, spreadsheets can easily contain errors. These errors may have severe consequences. In one case, for example, employees were overpaid by $700,000 [2]. Researchers have studied the kinds of errors that arise in spreadsheets [9] [17]. Examples include “accidental” errors, such as typos or mechanical errors with the mouse, that arise during the insertion or updating processes of cells. Another common kind of error is reasoning mistakes in formulas, which still can lead to quantitative errors. For example, an error total number can be caused by mistakenly choosing to use the wrong range in a SUM formula calculation. Other errors are “categorical,” when people put the wrong kind of data into a cell (e.g., a column mostly contains state names, except for one cell that has a zip code) [3]. Some errors are due to duplications, such as a row or individual cells that are incorrectly copied. Such structural errors can indicate flaws in the design or the layout of the spreadsheet.

Some of these errors can be found by existing approaches. Researchers have presented WYSIWYT (What You See Is What You Test), a testing methodology in research spreadsheet environment Forms/3 and commercial spreadsheet environment [6] [8] [10] [11] [12] [15]. This methodology allows users to make testing decisions and measure the quality of testing. If a value in a cell is correct, users can insert a checkmark in a box for each formula cell. If users choose not to do so, they can mark a question mark or leave the box empty. The measurement of this testing is shown by coloring the cell’s border from red (untested) to blue (tested). The overall testing progress is displayed by a percentage bar at the upper right corner in the spreadsheet. However, the drawback of manually tracking down errors is that it is tedious and time-consuming, and the WYSIWYT tool only partially automates the process of finding which cells to test. More importantly, WYSIWYT cannot find errors in textual cells such as “accidental” typographical errors or categorical errors.
Since 3-5% of string cells in spreadsheets contain errors, researchers have presented a specific approach for finding errors in string cells, especially for format errors [3]. This approach is called topes, which allows users to define formats, validate strings, and transform to another format. For example, users can define phone number formats by using a basic topes editor. The *isa* function, which returns a number between 0 and 1, is used to indicate the validity of parsing a string with the defined formats. When a spreadsheet is integrated with topes, topes will help users validate string cells with the defined formats. If a string cell violates the defined formats, topes will show a red triangle flag at the upper right corner of each error cell. Topes also shows a warning message if the string cell violates some constraints. Nevertheless, a major limitation of topes is that it cannot find some error categories, such as formula errors or duplicate cell errors.

Still another approach is to find errors based on unit analysis. UCheck is one algorithm for using this approach, which consists of two parts: a header inference system and a detecting errors system [20] [21]. The header inference system classifies cells in the spreadsheet into four categories: header, footer, data, and filler cell. These cells are assigned their roles by using heuristics such as content-based cell classification and footer-to-core expansion. When header cells are identified, UCheck assigns remaining cells to the data and footer cells. Another component of UCheck is the automatic detecting of errors. UCheck uses the unit inference technique. If its unit is not well-formed, the formula cell is marked as an error cell. Other formula cells which have references to the error cell are marked in a lighter color. The unit inference technique can detect some error categories in spreadsheets automatically such as formula reference errors or formula omission errors.

Researchers have implemented an approach to apply inter-class code smells for worksheets in a spreadsheet [28]. They define four inter-worksheet smells (e.g. *Inappropriate Intimacy, Feature Envy, Middle Man,* and *Shotgun Surgery*) to help users to understand the weak point in their spreadsheets. For example, when a formula
refers to cells in other worksheets, a spreadsheet may be difficult to understand and complicate. Data flow diagrams are used to illustrate these smells and weak points to users. When the data flow diagram indicates the cells that are a bad smell, users have to manually inspect the listed cells and judge their smelliness. This approach is concentrated on quality and design of a spreadsheet. In addition, this approach focuses only on bad smells among worksheets in a spreadsheet. Four inter-worksheet smells analyze formula cells and referred cells in other worksheets. Intra-worksheet smells and textual cells are not analyzed by this approach. All cells may lead to errors or weak points in a spreadsheet. It may be useful to apply more bad smells for a spreadsheet environment.

Some error categories still cannot be detected by any of these existing automatic approaches. Since Microsoft Excel allows users to copy formulas into adjacent cells easily, these duplicate formulas may contain duplication errors if the copied formula contains an error. Such errors cannot be detected by the existing approaches. In addition, existing approaches cannot detect some typographical errors, categorical errors, or duplicate cell errors. For instance, topes cannot detect duplicate cell errors and typographical errors. Many spreadsheets use the left-most column as a running number. User might type a duplicate number or omit any number. Due to these limitations, we will present an approach to detect both textual and formula errors in the next section.

2.4 Bad smells

The bad smells, which are terms introduced by Fowler, are an indication for underlying problems that might indicate errors in object-oriented programming [13]. These problems might require programmers to find errors or maintain the source code by refactoring. For instance, the Feature Envy bad smell is used to identify the misplacement of methods in Java programming [26]. A summary of other bad smells in object-oriented programming is shown below.
<table>
<thead>
<tr>
<th><strong>Bad smells</strong></th>
<th><strong>Description in the object-oriented environment</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Large Class</em></td>
<td>A class has a large number of members. Keeping these members consistent can be difficult.</td>
</tr>
<tr>
<td><em>Lazy Class</em></td>
<td>Classes do not contain many member.</td>
</tr>
<tr>
<td><em>Data Clumps</em></td>
<td>Sometimes several classes may contain similar members. Keeping them consistent can be difficult.</td>
</tr>
<tr>
<td><em>Long Method</em></td>
<td>Some members contain a very large amount of code, too much to fit on one screen and hard to keep right.</td>
</tr>
<tr>
<td><em>Long Message Chain</em></td>
<td>Some members are computed based on other members that are computed based on others, etc., and each dependency creates an opportunity to cause errors that propagate.</td>
</tr>
<tr>
<td><em>Long Parameter List</em></td>
<td>The large number of parameters might make a method too complex and hard to understand.</td>
</tr>
<tr>
<td><em>Feature Envy</em></td>
<td>Members may depend on data from other classes, and such dependencies can create opportunities for errors that can propagate.</td>
</tr>
<tr>
<td><em>Switch Statements</em></td>
<td>Complex conditional formulas might lead to errors due to the risk of duplication and complex logic.</td>
</tr>
<tr>
<td><em>Temporary Field</em></td>
<td>Members that serve no purpose except to store a temporary value really should be local variables if at all possible. Putting them in the class just creates opportunities for incorrect dependencies.</td>
</tr>
</tbody>
</table>

Figure 4. Examples of bad smells in object-oriented programming

Due to the similarity between object-oriented programming and a spreadsheet, the spreadsheet structures may have the same characteristics that cause underlying problems or errors in the spreadsheet. However, detecting errors by using bad smells has not been studied in a spreadsheet environment. Our approach will present how to
use bad smells detectors for finding some error categories in spreadsheets. We anticipate that detecting bad smells may reveal several types of errors in spreadsheets. There are different types of properties that can be considered in bad smells such as the number of rows, the number of formula parameters, etc. We describe our bad smell detectors in Section 3 and present our evaluation in Section 4.
3 Approaches

Detecting bad smells is our approach for finding errors in spreadsheets by analyzing structural characteristics. Our approach finds implicit relationships between each rectangular region’s columns, uses these functional dependencies to find keys in the rectangular region, and applies static analyses to detect bad smells in spreadsheets by analyzing rectangle’s characteristics, functional dependencies, best key, and formulas.

We decided to focus on Microsoft Excel, which is the most widely used spreadsheet environment. Our approach has been implemented by using Microsoft Visual Studio 2010 and the C# programming language. We also used COM interop for performing automated tasks in Excel such as open existing Excel spreadsheets, finding the current region, get cell’s address, etc. This Excel COM interop assembly can help us to work with Microsoft Excel seamlessly.

The process for detecting bad smells in spreadsheets is illustrated in Figure 5. Each step will be explained in the following sub-sections.
Figure 5. Process for detecting bad smells in spreadsheets

3.1 Extracting rectangular regions

In our approach, we define a region of each table as a rectangle which is surrounded by empty rows and empty columns. These empty rows and columns are used as separators between tables within spreadsheets. Since spreadsheets do not have any restrictions for creating tables, end users can create tables by using their own style, for instance, having multiple header rows, having several tables within spreadsheets, or having data cells without header information. Before detecting errors in spreadsheets, the boundaries of each table need to be identified as a rectangular region. Rectangular regions in our approach may contain contiguous cells which include empty cells, string cells, or formula cells. For example, one rectangular region may have header, data, and footer rows. Another region might contain only header and data rows.
Our algorithm for extracting the rectangular region is shown in Figure 6. To extract rectangular regions, our approach consists of two steps. First, we have to find at least one non-empty cell. The non-empty cell can contain any data types (e.g. string, numeric, date, or formula). Second, the neighboring cells of this non-empty cell are accumulated into a rectangle to identify their current region.

For instance, Figure 7 consists of three rectangular regions in a spreadsheet. Two of them contain merged cells (B2 and I4). When retrieving the first rectangle from the spreadsheet, the isFirstFindRectangle variable (line 1) is true. Our algorithm (Figure 6) reads the bottom-right cell (J9) from Excel’s xlCellTypeLastCell property. This property finds the last cell in the used range. The algorithm scans (line 2) from left to right (row by row) until it finds the first non-empty cell, which in this case is B2. The rectangle and foundRegionsSet are initialized to contain the current region of B2 (line 6 and 8). This CurrentRegion property provided by Excel returns the entire rectangular region of its non-empty cell, where the “region is a range bounded by any combination of blank rows and blank columns” [25].

After the rectangular region (B2:F6) was extracted, this algorithm (lines 10-24) searches other neighboring regions in the spreadsheet. The algorithm walks down the right-hand side of the rectangle (lines 10-16) and, for each row, scans to the right to find the next non-empty cell (line 12). Each such cell is used to identify another rectangle (lines 13-15), which in this case uncovers rectangle H4:J8. The process is then repeated by scanning across the bottom of the columns in the original rectangle B2:F6 (lines 18-24), discovering rectangle B8:F9. These two rectangles (to the right and below) are added to the nextRegionsQueue. The algorithm (Figure 6) is repeatedly called until the nextRegionsQueue is an empty queue. When the algorithm is called again, the isFirstFindRectangle will be false, and the algorithm will initialize the firstCell to point at each region in turn. As it proceeds through the rest of the algorithm, then it will be able to find additional rectangles located farther to the right or downward.
Algorithm Extract the rectangular region

Input: All cells on the worksheet range

Output: The rectangular region rectangle

1: if isFirstFindRectangle then
2: firstCell = Searching the first non empty cell across a row which starts from the last cell in the used range
3: else
4: firstCell = Searching the upper left cell in the extracted region nextRegionsQueue
5: end if
6: rectangle ← firstCell.CurrentRegion
7: lastCell ← rectangle.Item[the index number of the last cell]
8: foundRegionsSet.Add(rectangle)

9: for each row in rectangle do
10: rightMostCell ← rectangle.Item[the index number of the right most cell]
11: nextNonEmptyCell = Searching the non empty cell across a row which starts from rightMostCell
12: if !foundRegionsSet.Contains(nextNonEmptyCell.CurrentRegion) and nextRegionsQueue.Contains(nextNonEmptyCell.CurrentRegion)
13: then
14: nextRegionsQueue.Add(nextNonEmptyCell.CurrentRegion)
15: end if
16: end for
17: for each column in rectangle do
18: bottomCell ← rectangle.Item[the index number of the bottom cell]
19: nextNonEmptyCell = Searching the non empty cell down through a column which starts from bottomCell
20: if !foundRegionsSet.Contains(nextNonEmptyCell.CurrentRegion) and nextRegionsQueue.Contains(nextNonEmptyCell.CurrentRegion)
21: then
22: nextRegionsQueue.Add(nextNonEmptyCell.CurrentRegion)
23: end if
24: end for
25: return rectangle

Figure 6. Algorithm for extracting the rectangular region
Figure 7. An example of multiple rectangular regions in the spreadsheet [14]

3.2 Pre-processing step

Before finding implicit relationships between columns, our approach requires a pre-processing step to identify header rows in the rectangular region. Then our approach uses columns in the header row for finding functional dependencies. An overview of pre-processing is shown in Figure 8.

![Diagram](image)

Figure 8. An overview of pre-processing step
Each cell in a spreadsheet may be either a formula or a textual cell. Textual cells, which are string, number, or date, cannot refer to other cells. In contrast, formula cells can refer to (“reference”) other cells, potentially in other rows, as input variables for calculating the result. If any rows is not referenced, then it may be a header, data, or footer. To identify header rows, data rows and footer rows, our approach uses two row sets, which are referenced rows and unreferenced rows sets, in the pre-processing step.

For instance, a rectangular region shown in Figure 9 consists of string, formula, numeric, and empty cells. After parsing formula cells, two row sets are identified by the algorithm in lines 1-7 of Figure 10. The referenced set contains 3, 4, 5, 6, 7, 8, 9, and 10, since these are referred to by some formulas somewhere. The unreferenced set contains 1, 2, and 11. Then we extract the horizontal footer by analyzing each row below the last referenced row. This identifies row 11 as a footer row. Footers also can be recognized based on a whitelist of labels: lines 13-23 in Figure 11 (below) also extract row 10 based on this whitelist (which recognizes “average”).

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>ACME TOYS: WEEK 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Jones</td>
<td>13</td>
<td>total sales</td>
<td>8.25</td>
<td>107.25</td>
<td>25</td>
<td>=SUM(E3:F3)</td>
</tr>
<tr>
<td>4</td>
<td>Smith</td>
<td>15</td>
<td></td>
<td>7.5</td>
<td>112.5</td>
<td>22</td>
<td>=SUM(E4:F4)</td>
</tr>
<tr>
<td>5</td>
<td>Rogers</td>
<td>20</td>
<td></td>
<td>10</td>
<td>200</td>
<td>23</td>
<td>=SUM(E5:F5)</td>
</tr>
<tr>
<td>6</td>
<td>Harris</td>
<td>23</td>
<td></td>
<td>9.25</td>
<td>212.75</td>
<td>33.5</td>
<td>=SUM(E6:F6)</td>
</tr>
<tr>
<td>7</td>
<td>Adams</td>
<td>42</td>
<td></td>
<td>11.25</td>
<td>472.5</td>
<td>39</td>
<td>=SUM(E7:F7)</td>
</tr>
<tr>
<td>8</td>
<td>Stevens</td>
<td>44</td>
<td></td>
<td>10.75</td>
<td>473</td>
<td>60</td>
<td>=SUM(E8:F8)</td>
</tr>
<tr>
<td>9</td>
<td>Green</td>
<td>48</td>
<td></td>
<td>9.85</td>
<td>472.8</td>
<td>62.5</td>
<td>=SUM(E9:F9)</td>
</tr>
<tr>
<td>10</td>
<td>Max</td>
<td>=MAX(B3:B9)</td>
<td>=MAX(C3:C9)</td>
<td>=MAX(D3:D9)</td>
<td>=MAX(E3:E9)</td>
<td>=MAX(F3:F9)</td>
<td>=MAX(G3:G9)</td>
</tr>
<tr>
<td>11</td>
<td>Average</td>
<td>=AVERAGE(B3:B9)</td>
<td>=AVERAGE(C3:C9)</td>
<td>=AVERAGE(D3:D9)</td>
<td>=AVERAGE(E3:E9)</td>
<td>=AVERAGE(F3:F9)</td>
<td>=AVERAGE(G3:G9)</td>
</tr>
</tbody>
</table>

Figure 9. An example of a rectangular region [14]
After these two rows are identified as the footer rows, these rows will no longer be considered in the referenced and unreferenced sets. The algorithms in Figures 12 and 13 (below) then identify the header rows. Since the referenced set contains at least one row, the algorithm begins reading upward above the first referenced row. The algorithm notes differences in data types—that row 2 does not contain numeric cells, unlike row 3. Therefore, our approach can identify row 2 as the maximum number of the header row. Row 1 will be included in the header row as well. The remaining rows, 3-9, are identified as data rows within the rectangular region.

As with other parts of our approach, the algorithms outlined above incorporate many heuristics not guaranteed to work perfectly in every case. Section 4 will describe our empirical results showing that our approach generally works fairly well for many rectangular regions, after the subsections below discuss our preprocessing in detail and subsequent sections present our other analyses.

3.2.1 Row Sets

The first step of preprocessing is parsing formula cells and constructing the referenced rows set (lines 1-7 in Figure 10). We use the Excel DirectPrecedent property to find all direct precedent cells of each formula cell (i.e. cells used by the cell’s formula, if any). Referenced row numbers are added to the referenced rows set. Remaining rows, which are not used by formula cells, are grouped together as the unreferenced rows set.
Algorithm Pre-processing

Input: The rectangular region rectRegion
Output: The processed rectangle object R

1: referredRowSet = new Set<int>()
2: GetCellProperties(R, rectRegion)
3:
4: for each offsetRow in R.ReferredRow do
5:   referredRowSet.Add(offsetRow)
6: end for
7: unreferredRowSet ← R.Rows \ referredRowSet
8: GetHorizontalFooterRow(referredRowSet, unreferredRowSet, R.Rows)
9:
10: if referredRowSet is not an empty set then
11:   while unreferredRowSet.Max() > referredRowSet.Min() and
12:      unreferredRowSet is not an empty set do
13:     referredRowSet.Add(unreferredRowSet)
14:     unreferredRowSet.Remove(unreferredRowSet.Max())
15:   end while
16: else
17:   GetHeaderRow(referredRowSet, unreferredRowSet, R)
18: end if
19:
20: nonEmptyColSet = GetNonEmptyCol(rectRegion)
21: for each row in R.Rows do
22:   if row \ R.headerRowSet and row \ R.footerRowSet and row is not
23:      a subheader row then
24:     R.DataRowList.Add(rectRegion[row])
25:   end if
26: for each col in nonEmptyColSet do
27:   completedDataRow.Add(rectRegion[row][col])
28: end for
29: R.CompletedDataRowList.Add(completedDataRow)
30: end for
31: return R

Figure 10. Algorithm for pre-processing step
3.2.2 Extract footer rows

The second preprocessing step (Figure 11) is indicating footer rows. The algorithm is called \textit{GetHorizontalFooterRow}. If the maximum referenced row is less than the total number of rows, the rectangular region may contain footer rows. Lines 4-9 in Figure 11 check each row past the maximum referenced row, to see if each row contains a formula cell and the referenced rows set contains its referenced rows. If this condition is true, this row will be added to the footer rows set and removed from the unreferenced rows set. In addition, the algorithm recognizes that users may use aggregate formulas. Lines 13-23 in Figure 11 recognize such rows if they have numeric cells and use a string from our hardcoded whitelist (e.g. sum, total, average, etc.) to label the row.
3.2.3 Extract header rows

The third preprocessing step is extracting header rows (lines 10-18 in Figure 10). Header rows are identified by analyzing the unreferenced and referenced rows set. An unreferenced row above referenced rows is categorized as the header row if the unreferenced set contains only one row (Figure 12). When the unreferenced set
contains multiple rows, algorithms in Figures 13-14 are used to analyzed each unreferenced row above referenced rows.

---

**Algorithm** GetHeaderRow

**Input:** The processed rectangle object R, referred rows referredRowSet, and unreferred rows unreferredRowSet

**Output:** Header rows headerRowSet

1. headerRowSet = new Set<int>()
2. if referredRowSet is not an empty set then
3.   if unreferredRowSet.Count = 1 then
4.     headerRowSet.Add(unreferredRowSet)
5.   else
6.     headerRowSet = ComputeHeaderRowsByNonEmpty(referredRowSet, unreferredRowSet)
7.   end if
8. 
9. else
10. if unreferredRowSet.Count = 1 then
11.     headerRowSet.Add(unreferredRowSet)
12. else
13.     headerRowSet = ComputeHeaderRowsByEmpty(referredRowSet, unreferredRowSet)
14. end if
15. end if
16. return headerRowSet

---

Figure 12. Algorithm for extracting header rows. (The functions invoked in lines 6 and 13 are defined in Figures below.)

A algorithm in Figure 13 is called if the referenced set is not an empty set. Header rows are determined by comparing rows based on cell type. This algorithm analyzes the maximum row as the header row if it does not contain a numeric cell and has a different cell’s type from the minimum row in the referredRowSet (lines 6-7); in this case, any other unreferenced rows above this are also considered as the header rows and removed from the unreferenced set (lines 8-9). (When comparing an empty cell with any cells, these cells are considered as an identical type.) Otherwise, when
the last unreferenced row (above the referenced rows) consists of textual cells, this and upward unreferenced rows are also considered as the header rows and removed from the unreferenced set (lines 11-14). If neither of the two cases above are met, the algorithm must iterate \textit{maxDoubtRow} (the last unreferenced row above the referenced cells) upward toward the first row of the region. At any point in this iteration, if the header set is not an empty set, the algorithm compares the current \textit{maxDoubtRow} with the maximum row in the header set. When a similarity is found, the unreferenced set is included in the header set (lines 16-22). Otherwise, the maximum row is added to the referenced set and removed from the \textit{unreferencedRowSet} (lines 20-21). This iteration continues until the \textit{unreferencedRowSet} is an empty set.

A different header-identification algorithm is used when the referenced set is empty (typically due to textual cells). Figure 14 shows an algorithm for finding the header row in the unreferenced set. The minimum row in the \textit{unreferencedRowSet} is assigned to the \textit{minDoubtRow} and removed from the set (lines 3-4). If the \textit{minDoubtRow} is not an numeric cell and the \textit{unreferencedRowSet} is an empty set, this row is added to the \textit{headerRowSet} (lines 5-8). When the \textit{minDoubtRow} is an numeric cell, this step is done by removing rows in the \textit{unreferencedRowSet} (line 10). However, if the \textit{unreferencedRowSet} is not an empty cell, the algorithm compares the minimum row with the \textit{minDoubtRow} with the minimum row in the \textit{unreferencedRowSet} (line 14). When a difference is found, the \textit{minDoubtRow} is included in the header set and removed from the \textit{unreferencedRowSet} (lines 15-17). If the \textit{headerRowSet} is an empty set, the \textit{minDoubtRow} is added to the \textit{headerRowSet} (lines 20-21). The algorithm compares the \textit{minDoubtRow} with the maximum row in the \textit{headerRowSet} after a similarity is found and the \textit{headerRowSet} is not an empty set (lines 23-27). If the \textit{minDoubtRow} has an identical form to the maximum row, the \textit{minDoubtRow} is added to the \textit{headerRowSet}. Otherwise, rows in the \textit{unreferencedRowSet} are removed. The algorithm in Figure 14 is repeatedly done until the \textit{unreferencedRowSet} is an empty set.
Algorithm ComputeHeaderRowsByNonEmpty

**Input:** The processed rectangle object $R$, referred rows $\textit{referredRowSet}$, and unreferred rows $\textit{unreferredRowSet}$

**Output:** Header rows $\textit{headerRowSet}$

1. $\textit{headerRowSet} = \text{new Set<\text{int}>}()$
2. while $\textit{unreferredRowSet}$ is not an empty set do
3.   $\textit{minDataRow} \leftarrow \textit{referredRowSet}.\text{Min}()$
4.   $\textit{maxDoubtRow} \leftarrow \textit{unreferredRowSet}.\text{Max}()$
5.   $\textit{containsNumCell} = \text{ContainsNumericCell}(\textit{minDataRow}, \textit{maxDoubtRow})$
6.   $\textit{isIdenticalForm} = \text{HasIdenticalForm}(\textit{minDataRow}, \textit{maxDoubtRow})$
7.   if $\textit{!containsNumCell}$ and $\textit{isIdenticalForm}$ then
8.     $\textit{headerRowSet}.\text{Add}(\textit{unreferredRowSet})$
9.     $\textit{unreferredRowSet}.\text{Clear}()$
10. else if $\textit{!containsNumCell}$ and $\textit{isIdenticalForm}$ then
11.     if $\textit{headerRowSet}$ is an empty set then
12.        $\textit{headerRowSet}.\text{Add}(\textit{unreferredRowSet})$
13.        $\textit{unreferredRowSet}.\text{Clear}()$
14.     else
15.       if $\text{HasIdenticalForm}(\textit{maxDoubtRow}, \textit{headerRowSet}.\text{Max}())$ then
16.          $\textit{headerRowSet}.\text{Add}(\textit{unreferredRowSet})$
17.          $\textit{unreferredRowSet}.\text{Clear}()$
18.       else
19.         $\textit{referredRowSet}.\text{Add}(\textit{maxDoubtRow})$
20.         $\textit{unreferredRowSet}.\text{Remove}(\textit{maxDoubtRow})$
21.       end if
22.     end if
23. end if
24. else if $\textit{containsNumCell} = \text{true}$ then
25.    $\textit{referredRowSet}.\text{Add}(\textit{maxDoubtRow})$
26.    $\textit{unreferredRowSet}.\text{Remove}(\textit{maxDoubtRow})$
27. end if
28. end while
29. return $\textit{headerRowSet}$

Figure 13. Algorithm for computing non-empty referenced rows
Algorithm ComputeHeaderRowsByEmpty

Input: The processed rectangle object \( R \), referred rows \( referredRowSet \), and unrefereed rows \( unreferredRowSet \)

Output: Header rows \( headerRowSet \)

1: \( headerRowSet = new Set<\text{int}>() \)
2: while \( unreferredRowSet \) is not an empty set do
3: \( \text{minDoubtRow} \leftarrow \text{unreferredRowSet.Min}() \)
4: \( \text{unreferredRowSet}.\text{Remove}(<\text{minDoubtRow}>) \)
5: \( \text{containsNumCell} = \text{ContainsNumericCell}(<\text{minDoubtRow}>) \)
6: if \( \text{unreferredRowSet} \) is an empty set then
7: if \( \text{containsNumCell} \) then
8: \( \text{headerRowSet}.\text{Add}(<\text{minDoubtRow}>) \)
9: else
10: \( \text{unreferredRowSet}.\text{Clear}() \)
11: end if
12: end if
13: else
14: \( \text{isIdenticalForm} = \text{HasIdenticalForm}(<\text{minDoubtRow}, \text{unreferredRowSet.Min}()>)) \)
15: if \( \text{!containsNumCell} \text{ and } \text{!isIdenticalForm} \) then
16: \( \text{headerRowSet}.\text{Add}(<\text{minDoubtRow}>) \)
17: \( \text{unreferredRowSet}.\text{Clear}() \)
18: end if
19: else if \( \text{!containsNumCell} \text{ and } \text{isIdenticalForm} \) then
20: if \( \text{headerRowSet} \) is an empty set then
21: \( \text{headerRowSet}.\text{Add}(<\text{minDoubtRow}>) \)
22: else
23: if \( \text{HasIdenticalForm}(<\text{minDoubtRow}, \text{headerRowSet.Max}()>)) \)
24: then
25: \( \text{headerRowSet}.\text{Add}(<\text{minDoubtRow}>) \)
26: else
27: \( \text{unreferredRowSet}.\text{Clear}() \)
28: end if
29: end if
30: else if \( \text{containsNumCell} \) then
31: \( \text{unreferredRowSet}.\text{Clear}() \)
32: end if
33: end if
34: end while
35: return \( \text{headerRowSet} \)

Figure 14. Algorithm for computing empty referenced rows
3.2.4 Extract data rows

Finally, rows not in the header and footer sets are identified as data rows. The completed data rows, which contain non-empty cells for every column in each row, are extracted for further analysis (in subsequent sections). For this inclusion rule, we consider only cells in mostly-non-empty columns. Mostly-non-empty columns are any columns that contain non-empty cells in more than half of the total rows. Completed data rows can help to improve the performance of an algorithm for finding functional dependencies such as the execution time or the correctness of functional dependencies. This step is shown in lines 20-30 of Figure 10. (An alternative approach, instead of using this mostly-non-empty inclusion criterion, could be to include all rows, but then flag those with missing values as a form of trivial smell detection. We wanted to focus on testing more sophisticated data detectors, so we chose to simply apply the criterion above.)

3.3 Finding functional dependencies

We improved the FUN algorithm by removing any redundant functional dependencies, thereby reducing the set of functional dependencies to its minimal equivalent form. (This is straightforward but helps to improve performance in the computation that follows.) When each level of FUN found function dependencies, an algorithm in Figure 15 analyzes each functional dependency by comparing a closure of a left-hand side with a right-hand side. If the closure of the left-hand side contains all attributes of the right-hand side, this functional dependency is a redundancy.
3.4 Finding keys and the plausible key

In some rectangular regions we can extract multiple keys which comprise a set of columns. Selection of a primary key is arbitrary. We use heuristics to select a primary key that seems conceptually plausible as a reasonable choice. First, multiple key columns are more complex to insert data in the table. These key columns may also be hard to check for correctness. Consequently, the least number of key columns may be more convenient and useful than a large number of key columns. Second, the left-most column might often serve as a de facto primary key in spreadsheets [3]. Third, short numeric or easy-to-type values might be preferred over longer, complex strings.

Therefore, we implemented an additional algorithm for choosing the primary key based on the number of columns, the distance, the length, and the complexity of
typing the keys. The first criterion is the number of columns. If we found only one key which has the least number of columns in a set of keys, then this key is chosen to be the primary key. If several key sets are tied for smallest number of columns, the distance, the length and the complexity criteria are used to decide which key is preferable. For instance, two sets of keys, which are \{PN, EN\} and \{JC, H\}, are found by using functional dependencies in Section 3.3. In the case of such a tie, the shortest length criterion is used to find the primary key. \{PN, EN\} is chosen to be the primary key because the length of values in these columns is less than the other’s length.

Our algorithm is not guaranteed to extract the best key in every rectangular region. Instead, it relies on heuristics, as in an existing system developed by Cunha et al. [29]. As in that existing work, our heuristics are aimed at minimizing the number of columns in the key, the distance from the rectangle’s left edge, and the length and the complexity of typing the keys. However, our algorithm does not several other heuristics used in that system, such as those based on header semantics, nor the ratio between the functional dependency’s left-hand size and the right-hand size. In practice, we generally have found that our simpler heuristics work well most of the time at selecting a primary key as a plausible key, but using more sophisticated heuristics could offer an opportunity for improvement in the future.

### 3.5 Detecting bad smells

Each bad smell can be detected by a different static analysis. For example, some static analyses may be useful for detecting accidental (typo) errors. Other analyses might be useful for detecting formula errors. The summary of bad smells used in our approach is shown in Figure 16.
<table>
<thead>
<tr>
<th><strong>Bad smells</strong></th>
<th><strong>Description in the spreadsheet environment</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Large Class</strong></td>
<td>Tables may have a large numbers of columns. Sometimes these columns are only used occasionally. Large tables can also suffer from too many columns that require effort to keep consistent.</td>
</tr>
<tr>
<td><strong>Lazy Class</strong></td>
<td>Tables might not contain much useful information such as a lot of non-key columns or number of rows.</td>
</tr>
<tr>
<td><strong>Data Clumps</strong></td>
<td>Several tables may contain similar columns in spreadsheets. Users might forget to maintain data in these columns.</td>
</tr>
<tr>
<td><strong>Long Member</strong></td>
<td>Some columns may contain very long text which might cause accidental errors such as typographical error.</td>
</tr>
<tr>
<td><strong>Long Message Chain</strong></td>
<td>Some formula cells might rely on other cells in the spreadsheet. Our analyses count the number of cells which affect each formula cell. Dependencies might cause propagating errors.</td>
</tr>
<tr>
<td><strong>Long Parameter List</strong></td>
<td>The large number of parameters might make formulas complex and hard to understand. Our analyses check the number of arguments in each formula cell.</td>
</tr>
<tr>
<td><strong>Feature Envy</strong></td>
<td>Formula cells may depend on data from other tables in the spreadsheet. Our analyses trace the number of tables which is referenced by formula cells. Dependencies might cause errors.</td>
</tr>
<tr>
<td><strong>Switch Statements</strong></td>
<td>Complex lookup or conditional formulas might cause errors. Our analyses check the number of lookup and if-functions in each formula cell.</td>
</tr>
<tr>
<td><strong>Temporary Field</strong></td>
<td>Spreadsheets may contain hidden formulas or zero-width formula arguments. Such cells might be hard for people to check for correctness. Our analyses check the total number of these formulas in each table.</td>
</tr>
</tbody>
</table>
| **Duplicated Cell**  
*based on object-oriented idea of*  
"Duplicated Code"  
*but applied to data* | Spreadsheets may contain duplicate keys such as running numbers or identification numbers. Sometimes keys are not identical, but non-key columns are identical, potentially indicating duplication errors. |

Figure 16. Bad smells are used to analyze errors in spreadsheets

Our bad smell detectors do not detect errors. Instead, they (generally) compute a number indicating how bad a particular rectangular region looks. This number is compared to a threshold (outside of the detector itself) to determine whether the user should be invited to manually check that region for errors. This is similar to the approach of topes [3]: the responsibility of the analysis is only to return a number indicating possible validity, and a tool surrounding the analysis (e.g., a tope) determines whether to notify the user.

We recognize that even though these bad smell detectors are similar to detectors that work well for object-oriented programmers, maybe they will not work as well for spreadsheets. We will test them in Section 4, after we describe them in detail below.

### 3.5.1 Bad smell detectors that just count rows and columns

*Large Class, Lazy Class, and Data Clumps* analyze the structure of tables in spreadsheets. The number of columns are analyzed by *Large Class Detector*, while the number of rows and non-key columns are analyzed by *Lazy Class Detector*. We implemented *Lazy Class Detector* to counts the number of row and non-key columns separately because either non-key columns or rows may lead to errors. Non-key columns analysis is done by removing key attributes from the rectangle columns set. Then we count the remaining attributes in the rectangle columns set.
For example, a rectangular region has seven columns (PN, PA, EN, EA, JC, CH, and H) and 21 rows. Our approach in Section 3.4 can extract two key columns which are PN and EN. We can analyze this rectangular region by using Large Class and Lazy Class Detectors. The large class detector analyzes the number of columns and returns seven which include both key and non-key columns. The lazy class detector analyzes the number of non-key columns and the number of rows. These detectors return 5 and 21 respectively.

**Data Clumps Detector**, which is shown in Figure 17, analyzes duplicate columns among tables in spreadsheets. This detector computes the frequency of columns in the lowest-level header row. Only non-empty columns are included in the frequency list. For example, the frequency list of B2:F6 range in Figure 7 contains 4 columns which are week 1, week 2, week 3, and week 4. The first column is not included in the list due to an empty cell. Data Clumps Detector finds the most frequent column in each rectangular region. Finally, Data Clumps Detector shows the most frequent column overall.

To exemplify, Figure 7 shows two valid rectangular regions in the spreadsheet. Our approach for finding header rows can extract two header rows from B2:F6 and H4:J8 ranges. These rows are {<0>, week 1, week 2, week 3, week 4} and {<0>, <1>, gross pay}. Empty cells are replaced by their column’s index. All non-empty cells, which are week 1, week 2, week 3, week 4, and gross pay, are added to the frequency list. Since two header rows do not contain a duplicate cell, Data Clumps Detector will return the one which is in the maximum number of tables.
3.5.2 Bad smell detectors that rely on simple formula analysis

The next detector is Long Member. Long Member Detector checks the length of each cell in the rectangular region. This detector computes the average length of cells in each rectangular region. The algorithm is shown in Figure 18.

For example, the rectangular region in Figure 9 has four R1C1 formula cells which are SUM(RC[-2]:RC[-1]), MAX(R[-7]:R[-1]C), MAX(RC[-5]:RC[-1]), and
AVERAGE(R[-8]C:R[-2]C). The long member detector is used to count the number of characters in each data cell in the rectangular region. If the cell contains a formula, this detector will count the numbers of characters in A1 formula. Finally, *Long Member Detector* will show the average number of characters in all data cells which equals four.

### 3.5.3 Bad smell detectors that use complicated formula analysis

Four static analyses are required to parse formulas before detecting bad smells. These detectors are *Long Message Chain*, *Long Parameter List*, *Feature Envy*, and *Switch Statements*. For example, *Long Message Chain Detector* analyzes the path of computations. To identifying its precedent cells, each formula is required to parse before detecting bad smells. After that, *Long Message Chain Detector* computes the average number of precedent cells. The algorithm is shown in Figure 19.

#### Algorithm *Long Message Chain Detector*

**Input:** Each formula cell and its number of precedent cells *fxPrecedentDict*

**Output:** The average length of precedent cells in the rectangular region

1. `precedentList = new List<int>()`
2. `for each fx in fxPrecedentDict do`
3. `precedentList.Add(fx.Precedents)`
4. `end for`
5. `return precedentList.Average()`

![figure19](image)

Figure 19. Algorithm for *Long Message Chain Detector*

To exemplify, the formula cell G11 in Figure 9 is AVERAGE(G3:G9). This cell has six precedent cells that are G3, G4, G5, G6, G7, G8, and G9. These cells also have their own precedent cells (e.g. G3 has two precedent cells). The total number of precedent cells equals 21, providing many place where errors might creep in.
*Long Parameter List Detector* counts the number of arguments used in formulas (Figure 20). This detector also requires a formula parser. The grammar parser analyzes formulas in A1 format and can extract only explicit arguments. For instance, the formula cell B10 in Figure 9 is MAX(B3:B9). The grammar parser can extract two explicit arguments which are B3 and B9. The total number of arguments in Figure 9 equals 38.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th><em>Long Parameter List Detector</em></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong></td>
<td>A1 formula cells ( A1Formulas )</td>
</tr>
<tr>
<td><strong>Output:</strong></td>
<td>The total number of parameters in the rectangular region</td>
</tr>
<tr>
<td>1. ( paramList = new \text{List}&lt;\text{int}&gt;() )</td>
<td></td>
</tr>
<tr>
<td>2. \textbf{for each} ( fx ) \textbf{in} ( A1Formulas ) \textbf{do}</td>
<td></td>
</tr>
<tr>
<td>3. ( paramList.Add(ExcelFormula.FormulaParser(fx).Count) )</td>
<td></td>
</tr>
<tr>
<td>4. \textbf{end for}</td>
<td></td>
</tr>
<tr>
<td>5. \textbf{return} ( paramList.Sum() )</td>
<td></td>
</tr>
</tbody>
</table>

Figure 20. Algorithm for *Long Parameter List Detector*

*Feature Envy Detector* analyzes formulas that depend on other rectangular regions. After precedent cells are identified, we compute the region of each precedent cell. If the regions are not equal to their referencing rectangular region, the formula cell is dependent on other table. *Feature Envy Detector*, which is shown in Figure 21, computes the total number of dependent regions. For instance, a formula cell C8 in Figure 7 is SUM(C4:C6). This formula has three precedent cells that are C4, C5, and C6. The region of these precedent cells is B2:F6 which is another rectangular region in the spreadsheet. *Feature Envy Detector* will show one dependent region.
Algorithm Feature Envy Detector

Input: Each formula cell and its number of other dependent rectangle \( \text{fxDependentDict} \)

Output: The total number of dependent rectangle in the rectangular region

1: \( \text{dependentList} = \text{new List<int>()} \)
2: for each \( \text{fx} \) in \( \text{fxDependentDict} \) do
3: \( \text{dependentList.Add(fxDEPENDENT)} \)
4: end for
5: return \( \text{dependentList.Sum()} \)

Figure 21. Algorithm for Feature Envy Detector

Switch Statements Detector checks if formulas contain lookup functions or if-statements. Lookup functions, which are checked by this detector, are lookup, choose, index, match, and offset. We have implemented regular expressions for parsing formulas. Switch statements detector, which is shown in Figure 22, computes the total number of lookup functions or if-statements.

Algorithm Switch Statements Detector

Input: All formula cells \( AlFormulas \)

Output: The total number of lookup and if-functions in the rectangular region

1: \( \text{lookupList} = \text{new List<int>()} \)
2: for each \( \text{fx} \) in \( \text{AlFormulas} \) do
3: \( \text{fxUpper = fx.ToUpper()} \)
4: if \( \text{fxUpper} \) contains IF, LOOKUP, CHOOSE, INDEX, MATCH, or OFFSET then
5: \( \text{LookupExpr lookupRegEx = new LookupExpr()} \)
6: \( \text{lookupList.Add(lookupRegEx.Matches(fxUpper).Count)} \)
7: end if
8: end for
9: return \( \text{lookupList.Sum()} \)

Figure 22. Algorithm for Switch Statements Detector
3.5.4 Bad smell detectors for finding hidden or duplicate cells

For each formula, Temporary Field Detector checks hidden and zero-width formulas in rectangular regions. We have defined a rule to parse zero-width formulas by the grammar parser. Hidden formulas are checked by the Excel FormulaHidden property. This detector computes the total number of hidden and zero-width formulas. The algorithm is shown in Figure 23.

```
Algorithm Temporary Field Detector
Input: A1 formula cells A1Formulas
Output: The total number of hidden and zero-width formulas in the rectangular region
1:   tmpField = 0
2:   func0 = zero-width formula grammar
3:   for each fx in A1Formulas do
4:     if fx.FormulaHidden = true then
5:       tmpField = tmpField + 1
6:       continue
7:     else if ExcelFormula.FormulaParser(fx, func0).Count > 0 then
8:       tmpField = tmpField + 1
9:     end if
10:   end for
11: return tmpField
```

Figure 23. Algorithm for Temporary Field Detector

Finally, we have implemented Duplicated Cells Detector for finding duplicate cells error in the key and non-key columns. This detector compares the number of distinct values between completed data rows and all data rows which include both complete and incomplete data rows. If a difference is found, the cell columns might contain a duplicate cells error. The algorithm is shown in Figure 24.
Algorithm  Duplicated Cell Detector

Input: header cells hCells, data cells dCells, and key column K
Output: The predicted result

1: numEmptyKey = 0
2: distinctKeys = CountDistinct(hCells, dCells, K, outnumEmptyKey)
3: if numEmptyKey + distinctKeys ≠ dCells.Count then
   4: return true
5: else
6:   nonKeyCols ← hCells
7:   for each key in K do
8:     nonKeyCols.Remove(key)
9:   end for
10:  distinctKeys = CountDistinct(hCells, dCells, nonKeyCols, outnumEmptyKey)
11: if numEmptyKey + distinctKeys ≠ dCells.Count then
12:   return true
13: else
14:   return false
15: end if
16: end if

Figure 24. Algorithm for Duplicated Cell Detector
4 Evaluation

To evaluate these static analyses for detecting bad smells in spreadsheets, we tested our approach on spreadsheets from the existing EUSES Spreadsheet Corpus [14]. To summarize the result of our evaluation, we computed true positive and false position rates. True positive shows the fraction of invalid rectangles correctly predicted as errors. False positive shows the fraction of valid rectangles erroneously predicted as errors. We have found that six detectors can detect errors in spreadsheets fairly well and might be worthy of further investigation in future work.

4.1 Testing smell detectors

We tested 11 detectors on the database, computer science course, inventory, and science/engineering sections of the EUSES Spreadsheet Corpus [14]. These were selected to include spreadsheets with many text, number, and formula cells. We have manually reviewed the spreadsheets and found that they contain errors in both data and formula cells. We have categorized these errors into five groups which are defined as the real error in our approach. There might be other errors in the spreadsheets besides the ones that were obvious to us; we are just testing how well our approach can find the errors that we know about (true positives) without identifying too many cells that do not have obvious errors (false positives).

Data errors were divided into three groups which are categorical, duplication, and accidental errors. Categorical errors occurred when any cells contain data that are not defined by each column. For instance, a cell in a state columns contains a country. Duplication errors are any rows that contain the same value in the key column or other columns such as duplicate titles or duplicate running numbers. The remaining “accidental” errors are obvious typographical errors (e.g. string cells might start with an errant question mark).

Based on the characteristics of formula errors, we divided formula errors into two categories, duplication and quantitative errors. Duplication errors are any cells
that contain the same formula in R1C1 format. Quantitative errors included any remaining reasoning and accidental errors. Errors in reasoning appeared as using an obviously wrong formula. Such errors can also appear due to accidental errors when entering formulas during inserting or updating stages.

We anticipated that different detectors will find different errors. For some detectors, it was obvious which kinds of errors would most likely be found; for example, the *Duplicate Cell* detector clearly would be best at finding duplicated cells. Likewise, the *Long Parameter List* detector would likely be best at finding formula errors but of no use in finding data errors. In other cases, we had to guess what kinds of errors would most likely be found. The Figure below shows what errors were used to test each detector.

<table>
<thead>
<tr>
<th>Detectors</th>
<th>Accidental errors</th>
<th>Categorical errors</th>
<th>Duplicate cell errors</th>
<th>Duplicate formula errors</th>
<th>Quantitative formula errors</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Large Class</em></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td><em>Lazy Class</em> (tuple)*</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td><em>Lazy Class</em> (non-key)*</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td><em>Data Clumps</em></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td><em>Long Member</em></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td><em>Long Message Chain</em></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td><em>Long Parameter List</em></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td><em>Feature Envy</em></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td><em>Switch Statements</em></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
We tested our detectors by varying the thresholds for each bad smell in rectangular regions (i.e., how big the number returned by the detector had to be, before a cell would be flagged for a user to check it). We analyzed how the threshold affects each detector’s performance. In addition, our approach also uses the best threshold, which is computed by finding the maximum value between the true positives and the false positives, to predict whether any rectangular regions contain an error. For example, if the number of non-key columns is greater than or equal to five, a rectangular region is predicted as an error region.

We compute true and false positive rates based on whether each rectangular region is identified as containing an error, as shown below. This is appropriate because most of our smell detectors operate on a rectangle-level (just as most object-oriented smell detectors operate on the class-level). The true positive computes the fraction of correctly predicted erroneous rectangular regions. The false positive computes the fraction of incorrectly predicted rectangular regions. For example, if the rectangular region has errors and our detectors can predict correctly, we record the result as a true positive. Otherwise, we record the result as a false positive.

\[
\text{True positive} = \frac{\text{The number of incorrect rectangles predicted as containing errors}}{\text{The number of incorrect rectangles}}
\]

\[
\text{False positive} = \frac{\text{The number of correct rectangles predicted as containing errors}}{\text{The number of correct rectangles}}
\]
4.2 Results

We have generated graphs below showing the relationship between true positives and false positives as we vary each detector’s threshold. Each dot in the graph represents the result of using a certain value for the detector’s threshold. Graphs will show which detectors are useful for detecting errors in spreadsheets if dots tend to have a curve above where a diagonal line would be (which indicates the accuracy that would be expected by chance alone). The overall error rate was 12% of rectangular regions.

Figure 26. The result of Large Class Detector
Figure 27. The result of Lazy Class (tuple) Detector

Figure 28. The result of Lazy Class (non-key) Detector
Figure 29. The result of *Data Clumps Detector*

Figure 30. The result of *Long Member Detector*
Figure 31. The result of *Long Message Chain Detector*

Figure 32. The result of *Long Parameter List Detector*
Figure 33. The result of *Feature Envy Detector*

Figure 34. The result of *Switch Statements Detector*
Figure 35. The result of *Temporary Field Detector*

Figure 36. The result of *Duplicated Cell Detector*
According to the above results, we have found that six detectors have dots that
tend to bend in a curve above the diagonal line. These detectors are Large Class, Lazy
Class (tuple), Lazy Class (non-key), Long Message Chain, Long Parameter List, and
Duplicated Cell Detector. They show the moderately good results for detecting their
intended error categories.

In addition, we also considered whether it would be helpful to combine some
of these good detectors. We placed five detectors into eight groups which have two
detectors per group. We configured each detector with whatever threshold value
provided the greatest difference between true positive and false positive rates (i.e., the
greatest distance above the diagonal line in the Figures above). We then combined the
detectors pairwise within their groups using Boolean AND or OR operators. The result
in Figure 43 shows that two groups, which are a combination of Long Message Chain
and Long Parameter List Detector, have the approximate true positive equals 0.8-0.93
and the approximate false positive equals 0.15-0.34. For example, showing an alert to
a user when Large Class AND Lazy Class (tuple) exceeded their thresholds led to a
false positive rate of 0.25 and a true positive rate of 0.55 (dark square, Figure below).
Most of the other combinations provided even higher true positive rates, although with
higher false positive rates as well.
Overall, results for several detectors were particularly encouraging and suggest that the detectors could be worth investigating further in the future. *Lazy Class (tuple) Detector* was able to relatively accurately detect all error categories, which included both textual errors (e.g. categorical errors, accidental errors, or duplicate cell errors) and formula errors (e.g. duplicate formula and quantitative formula errors). Categorical errors and accidental errors were also detected by *Large Class* and *Lazy Class (non-key) Detector*. *Duplicate Cell Detector* was useful when any rows contained a duplicate key column or duplicate non-key columns. *Long Message Chain* and *Long Parameter List* worked fairly well to detect formula errors such as duplicate R1C1 formulas or syntax errors.

Our results also show that some detectors cannot find errors in spreadsheets well at all. These detectors are *Data Clumps, Long Member, and Temporary Field*
Detector. For example, Data Clumps Detectors demonstrated poor result because very few rectangular regions contained very identical columns in a spreadsheet, limiting its coverage of errors. Spreadsheets may contain a few hidden formulas, which likewise caused Temporary Field Detector shows poor result. Some detectors might just be irrelevant to spreadsheets even though they work well for object-oriented programmers. For example, most cell values were short, and most errors occurred in short-valued cells, so the Long Member Detector detected few errors due because few errors appeared in very long text.
5 Conclusion And Future Work

5.1 Conclusion

Our work is a first step toward a new approach for finding errors in spreadsheets, based on bad smell detectors. In our evaluation, we categorized errors in spreadsheets into five categories: categorical errors, accidental errors, duplicate cell errors, duplicate formula errors, and quantitative formula errors. Six detectors performed fairly well: Lazy Class (rows), Lazy Class (non-key), Large Class, Duplicated Cell, Long Message Chain, and Long Parameter List. Textual errors (i.e. categorical errors or accidental errors) were best detected by Large Class and Lazy (non-key) Class Detector. Formula errors (i.e. quantitative formula errors or duplicate formula errors) can be detected by Long Message Chain, and Long Parameter List Detector. In addition, Lazy Class Detector, which checks the number of rows, can detect all five error categories but at a fairly low accuracy. We conclude that these detectors could be useful for future work but will require improvements.

5.2 Future work

Future work could aim to increase the flexibility, specificity and accuracy of our approach. For example, in the current implementation of extracting rectangular regions, we decided to extract rectangular regions which are bound by empty rows and columns. Each row or column in the region cannot be empty. If the region contains at least one empty row or column, our algorithm cannot extract the entire region. These regions are extracted to two regions. Future work might improve flexibility by allowing regions that contain more blank values or are surrounded by non-blank space.

In addition, we must increase the accuracy of some bad smells detectors by improving the existing algorithms or adding the new algorithms. For instance, Lazy Class and Large Class Detectors show only a moderate rate of true positives. In addition, Feature Envy and Switch Statements Detector might be useful if we can increase the true positive rate. We may also implement the new bad smells detectors to
support more error categories in the future. In addition, we will need to check if normalizing smells (e.g. by computing proportions of rectangles, rather than counting rows) leads to different results.

Finally, we might improve specificity by improving our detectors so they can pinpoint errors inside rectangular regions. For example, detectors could be refined to analyze sub-regions, then intersected to highlight specific cells. Substantial work will be required to ensure accurate identification of errors, especially if the detectors are being allowed to identify larger numbers of errors to increase true positive rate.
Bibliography


