Seminar Paper

Smells and Units: An Overview of Selected Static Analysis Methods for Spreadsheets

Patrick Koch

Institute for Software Technology,
Graz University of Technology

Supervisor: Univ.-Prof. Dipl.-Ing. Dr. Franz Wotawa
Supervisor: Dipl.-Ing. Dr. Birgit Hofer

Graz, 2015-03-02

Abstract

The use of spreadsheets is the most popular form of end-user programming. As spreadsheets are in general not created and maintained by professional programmers, error rates are high. As a remedy, many approaches were proposed to avoid, find, and fix errors in spreadsheets. Although some literature on the current overall state of spreadsheet quality assurance (QA) is available, little work was published which gives an in-depth comparison of specific approaches. In this paper, we provide a more comprehensive survey of the proposed literature covering two chosen static spreadsheet QA techniques: The topics of spreadsheet smells and unit-checking of spreadsheets. We define the concept of spreadsheet smells and present a comprehensive catalogue containing each smell proposed in scientific literature. We highlight groups of related smells, remark on the similarities and interactions within these groups, and state how identified smells can be utilized. As a novel finding, we point out an identified gap within the currently available catalogue. With regard to unit-checking, we review the development of this principle as applied to spreadsheets. We identify and describe concepts commonly used by concerning approaches, elaborate on how those concepts can be expanded on, and name a view examples. Based on this review, we conclude on problems and provide suggestions for future work within both of the surveyed fields.

Keywords

Spreadsheets; Static Analysis; Spreadsheet Smells; Refactoring; Automatic Error Detection; Unit; Dimension; Debugging;

Table of Contents

1. Introduction

Spreadsheets play an important part in today's economy. In 2012, Panko and Port [2012] estimated that 60 % of the 90 million people using computers at work in the USA used spreadsheets and databases. Moreover, a considerable fraction of knowledge workers even considered their End User Computing application of choice as mission critical. Failure, unavailability, or incorrectness of such an application results in meaningful economic losses. However, the study also revealed that only 15 % of the personal working with spreadsheets and databases is made up of trained programmers. Poor spreadsheet quality and suggested error-rates of up to 90 % are the consequences.

As a countermeasure, there has been extensive research into improving QA for spreadsheet applications in the last decade. Various approaches to prevent, detect, and fix errors within spreadsheets have been proposed. In particular, numerous approaches were made to adopt common techniques for code quality improvement into the spreadsheet domain. Jannach *et al.* provided a comprehensive overview of noteworthy approaches to improve spreadsheet quality [2014].

Although considerable research has been spent on concrete approaches to improve spreadsheet quality assurance (QA), less attention has been paid to a more general perspective of the topic. In particular, research highlighting and comparing different approaches following a similar methodology is sparse. The result of such research can provide valuable information to derive best practices and identify gaps within the specific field. We perceive the field of static spreadsheet analysis, approaches which adapt and apply techniques of static code analysis to spreadsheet structures, to be of interest with that regard. Consequently, the aim of the present paper is to give an extensive overview and comparison of two specific static

analysis techniques for spreadsheets: spreadsheet smells and (label-based) unit- and dimension-checking of spreadsheets.

The remainder of this paper is as follows: In Section 2, we introduce the notion of spreadsheet smells. In daily life, if we perceive that something stinks, we are more suspicious of the quality of the object in question. Likewise, spreadsheet smells point out possible quality issues in spreadsheets. We present a complete catalogue of spreadsheet smells as they were proposed within the scientific community. For each smell within the catalogue, we provide detailed information based on a defined set of criteria. We highlight similarities and interactions between specific smells. We conclude our elaboration of spreadsheet smells in describing how the concept has been utilized throughout the present literature.

In Section 3, we summarize proposed efforts to adapt unit- and dimension-checking-techniques to the requirements of spreadsheet systems. We continue by describing the concepts commonly encountered within these proposed approaches. Lastly, we remark on how further concepts can be derived based on the notion of unit-checking of spreadsheets and provide concrete examples.

Section 4 concludes the paper and suggests topics for future work.

2. Spreadsheet Smells

The idea of spreadsheet smells is based on the notion of code smells, which was introduced in [Fowler, 1999]. In his work, Fowler promoted refactoring as a method to improve the quality of object-oriented code. However, no common guidelines as to when to refactor were readily available at that time. Therefore, Fowler proposed the utilization of a new concept for that purpose: code smells. According to Fowler's definition, a bad smell is a certain circumstance within object-oriented source code which suggests a certain refactoring. As such, a smell is defined by a specific anticipated flaw within the structure of object-oriented code. Duplicated code sections in unrelated classes are an example of such a flaw. In his work, Fowler proposed a list of 22 bad smells concerning object-oriented source code.

Spreadsheet smells are a recent development within the field of static spreadsheet analysis. As the name suggests, this method revolves around the conception and detection of bad smells within a spreadsheet. Cells and worksheets that contain a smell are either more likely to contain a fault or harder to maintain than non-smelly ones. Therefore, spreadsheet smells define a quantifiable measure of suspiciousness towards specific cells and worksheets.

Code smells do not directly indicate faulty code statements by themselves. Instead, code smells indicate code segments which are hard to comprehend, hard to maintain, or error-prone. Likewise, spreadsheet smells in general do not indicate faulty cells by themselves. They rather highlight localized quality-issues in a spreadsheet.

2.1 Known Spreadsheet Smells

In recent years, a number of different approaches have been made to adopt the notion of bad smells into the spreadsheet domain. In doing so, those approaches contributed to the set of bad spreadsheet smells known to the scientific community. In the following section, we present this collection of known spreadsheet smells. For each smell within the set, we discuss the following aspects:

- *Origin & Intent:* We name the origin and intent of the smell.
- *Target:* We define which parts of a spreadsheet can contain the smell.
- *Detection:* We describe how the smell can be detected.
- *Example:* We provide an example, based on an exemplary spreadsheet presented in Figure 2.1.
- *Cause:* We state possible causes of the smell.
- *Consequences:* We elaborate on which issues may be implied by the smell.
- *Alleviation:* We suggest ways to remove the smell.

To illustrate how each smell may affect a spreadsheet in practice, we provide an example spreadsheet containing an occurrence of each discussed smell in Figure 2.1. The example spreadsheet describes the internal processes of a basic warehouse company. Figure 2.1a depicts the *Sales* worksheet, aggregating product sales and revenues per period. Figure 2.1b illustrates the *Employees* worksheet, containing employee data and summarizing working hours. Lastly, Figure 2.1c presents the *Totals* worksheet, combining revenue and expenses to calculate total and periodic results as well as a productivity quota for the current period.

The following definitions are required to describe the detection processes of certain smells:

Cell	**C** represents a spreadsheet cell.
Worksheet	The type **W** represents a worksheet. It is defined as a set which contains all cells contained in a worksheet.

Spreadsheet	The type **S** represents a spreadsheet. It is defined as a set which contains all worksheets **W** contained in a spreadsheet.
Precedents	**P** denotes all the predecessors, of a specific formula. Predecessors are cells which a formula refers to. Thus, **P** is a function of type $\mathbf{C} \rightarrow \{\mathbf{C}\}$.
Connection	A tuple of cells (A, B) form a connection **K** if they adhere to the relation $A \in \mathbf{P}(B)$.
Connection Set	**KS** denotes the set which contains all connections of a spreadsheet.

2.1.1 Standard Deviation

Origin & Intent. Standard Deviation was first proposed by Cunha *et al.* in [2012b]. The smell is designed to detect statistical outliers within groups of numeric cells.

Target. Standard Deviation detects smelly numerical values. Thus, it can only be applied to cells which contain such values. Moreover, the smell is usually applied to input cells only. Numeric results of formula cells are ignored in the detection process.

Detection. Detection of the Standard Deviation smell relies on the statistical properties of a group of cells. To that end, groups of neighbouring cells in either column or row orientation are formed. For each occurring group, the normal distribution model of its contained numeric values is calculated. Based on this model, cells within the group are marked as smelly, if their cell value deviates by two standard deviations (95,4 %) from the calculated average of the group.

Example. An occurrence of the Standard Deviation smell can be seen in cell B3 of the *Sales* worksheet. This worksheet is depicted in Figure 2.1a. We calculated the average of the values within column B to be 9.21E8 and the standard deviation of the values in the column to be 2.90E8. Therefore, values within the column are expected to lie within the range [3.41E8,1.50E9]. However, cell B3 contains the value 1.08E3. This lies outside the expected range. Thus, cell B3 is marked as smelly.

	A	B	C	D	E	F	G	H
1	**id**	**reference**	**description**	**period**	**quantity**	**price**	**revenue**	**after tax**
2	5	1007993410	Product One	0	632	$19.99	=F2*E2	=F2*E2*0.9
3	5	1079	Product One	1	431	$19.99	=F3*E3	=F3*E3*0.9
4	5	1007993410	Product One	2	621	$19.99	=F4*E4	=F4*E4*0.9
5	5	1007993410	Product One	3	743	$19.99	=F5*E5	=F5*E5*0.9
6	6	1002394514	Product Two	o	79		=F6*E6	=F6*E6*0.9
7	6	1002394514	Product Two	1	113	$29.99	=F7*E7	=F7*E7*0.9
8	6	1002394514	Product Two	2	129	$29.99	=F8*E8	=F8*E8*0.9
9	6	1002394514	Productt Two	3	244	$29.99	=F9*E9	=F9*E9*0.9
10	7	1005237614	Product Three	0	1360	$19.99	=F10*E10	=F10*E10*0.9
11	7	1005237614	Product Three	1	1194	$19.99	=F11*E11	=F11*E11*0.9
12	7	1005237614	Product Three	2	1221	$18.99	=F12*E12	=F12*E12*0.9
13	7	1005237614	Product Three	3	1281	$19.99	=F13*E13	=F13*E13*0.8
14						**period**	=sumif(D2:D13,Totals!C1,G2:G13)	=sumif(D2:D13,Totals!C1,G2:G13)*0.9
15			**current period**	=Employees!B21		**total**	=sum(G2:G13)	=G15*0.9

(a) *Sales* worksheet.

	A	B	C	D
1	**name**	Anderson John	**occupation amount**	6
2	**salary**	12	**occupation status**	=if(D1<8,"minimum wage",if(D1<30,"part_time",if(D1<40,"full_time","over_time")))
3	**peroid**	**hours**	**salary**	
4	0	32	=B4*B2	
5	1	24	=B5*B2	
6	2	33	=B6*B2	
7				
8	**name**	Peterson Sarah	**occupation amount**	16
9	**salary**	$14.00	**occupation status**	=if(D8<8,"minimum wage",if(D8<30,"part_time",if(D8<40,"full_time","over_time")))
10	**period**	**hours**	**salary**	
11	2	64	=B11*B9	
12	3	70	=B12*B9	
13				
14	**name**	Craig Toby	**occupation amount**	38.5
15	**salary**	$15.00	**occupation status**	=if(D14<8,"minimum wage",if(D14<30,"part_time",if(D14<40,"full_time","over_time")))
16	**period**	**hours**	**salary**	
17	1	155	=B17*B15	
18	2	158	=B18*B15	
19	3	161	=B19*B15	
20				
21	**current period**	=Totals!C1	**period**	=sumif(A4:A19,B21,C4:C19)
22			**total**	=sum(B4:B6)*B2+sum(B11:B12)*B9+sum(B17:B19)*B15

(b) *Employees* worksheet.

	A	B	C
1	**Current period**	**current period**	3
2		**sum sales**	=Sales!G14
3		**sum sales after tax**	=Sales!H14
4		**sum employee costs**	=Employees!D21
5		**total**	=Sales!H14-Employees!D21
6			
7	**Total**	**sum sales**	=Sales!G15
8		**sum sales after tax**	=Sales!H15
9		**sum employee costs**	=Employees!D22
10			=Sales!H15-Employees!D22
11			
12	**Quota**	**num periods**	=count(unique(Sales!D2:D13))
13		**average result**	=C10/C12
14		**surplus / deficit**	= (Sales!H14 - Employees!D22) / ((Sales!H15-Employees!D22) / C12)

(c) *Totals* worksheet.

Figure 2.1: Warehouse example spreadsheet.

Cause. Standard Deviation indicates the occurrence of unexpected numeric cell values. Such values are usually introduced due to errors in the stage of data entry. Smelly numerical values also may be a result of ill-considered copy & paste operations. Furthermore, accidental alteration of cell values throughout the lifecycle of a spreadsheet can introduce deviating values. Poor spreadsheet layout may also cause the smell. For example, a row or column may contain multiple successive groups of related numerical values to be computed alongside. However, such groups do not necessary follow the same mathematical distribution. Without some form of boundary between such value groups, some values within the row or column may be marked as smelly as a result.

Consequences. The Standard Deviation smell affects numerical input cells. Spreadsheet users rely on the values provided by this type of cell to conduct various computations and statistic analyses. Therefore, a deviating input value will in most cases propagate throughout the spreadsheet and result in an error in at least one of the output values.

Alleviation. Correction of a cell which is tagged with the Standard Deviation smell depends on the circumstance which caused the cell to be smelly. In every case, the cell value should be examined at first. If a faulty cell value is detected, correcting it will usually also remove the smell. If the cell value was not faulty, or the cell is still indicated as smelly after correcting the value, structural considerations need to be applied. As described before, Standard Deviation may be detected due to successive groups of numeric values within the same row or column. In such cases, we suggest structural refactoring of the worksheet in question: Either introduce some form of boundary between the value groups, or split the groups into corresponding tables.

2.1.2 Empty Cell

Origin & Intent. The smell Empty Cell has its origins in [Cunha *et al.*, 2012b]. In their work, Cunha *et al.* proposed this smell to indicate cells which are empty, but occur within a context which suggests that the cell should contain a value.

Target. Empty Cell detects smelly empty cells. Consequently, it can only be applied to cells that do not contain any values, labels, formulas or errors.

Detection. This smell intends to detect empty cells occurring in a suspicious context. Such a context is described by a number of neighbouring cells that do not contain any other empty cells. Cunha *et al.* proposed to utilize windows of five cells for this purpose. During smell detection, for each row or column every possible window of 5 neighbouring cells is considered and verified whether it holds precisely one empty cell. If an empty cell only occurs within such groups, it is indicated as smelly.

Example. We can see, that cell F6 of the *Sales* worksheet, depicted in Figure 2.1a, is empty. Additionally, it is exclusively contained in 5-cell-neighbourhoods that do not contain any other empty cells. As a result, for this cell the Empty Cell smell is signalled. Notice that, for example, cells in row 14 also contain empty cells. However, those cells occur in neighbourhoods with other empty cells. Subsequently, they are not marked as smelly.

Cause. The Empty Cell smell indicates empty cells which are surrounded by non-empty cells. Empty cells within a table are usually introduced by mistake. During data entry or spreadsheet creation, sometimes a cell is overlooked by accident. An empty cell within a table may also be the result of an ill-considered copy & paste operation. Lastly, the empty cell may have been introduced by an accidental delete operation later within the lifecycle of the spreadsheet.

Consequences. Empty Cell indicates empty spots within the bulk of a worksheet. Cells within this area usually contain numeric input values or formulas. Those formulas either calculate final results or are themselves referenced by other formula cells. Thus, empty cells within those areas usually lead to missing or erroneous interim and final results.

Alleviation. In order to remove the Empty Cell smell from a cell, we suggest to check whether the cell in question should indeed be empty. If this is not the case, determine the missing content and insert it. Suggestions for the missing content may be automatically derived from the surrounding context and presented to the user.

2.1.3 Pattern Finder

Origin & Intent. The Pattern Finder smell was proposed by Cunha *et al.* in [2012b]. It can be seen as extension of the Empty Cell smell. Instead of focussing on empty cells, Pattern Finder attempts to detect more general deviations of expected cell types: for example, a label cell situated in a neighbourhood of cells containing numeric values.

Target. Pattern Finder attempts to detect unexpected deviations of occurring cell types within rows or columns of a worksheet. The focus thereby lies on the deviation itself, rather than the specific type. Thus, every cell within the active area of a worksheet is a potential target for the detection of the Pattern Finder smell.

Detection. The method of detecting the Pattern Finder smell is based on the methods that are applied to detect the Empty Cell smell. Detection of Pattern Finder relies on the inspection of the local neighbourhood of the cell in question. To that end, Cunha *et al.* proposed to form windows of 4 neighbouring cells for each row or column in the spreadsheet. For each of those windows, we need to check whether it contains exactly one cell containing a different type than the remainder of the group. If such a cell is detected, it is flagged as smelly.

Example. In the *Sales* worksheet, illustrated in Figure 2.1a, cell D6 contains the String "o". This cell thus has the cell type label. However, the surrounding cells in the same column contain number values. Therefore, D6 is indicated as smelly. It is likely that the cell should have contained the number 0 instead of the label "o": a typing error occurred.

Cause. Cells which are flagged with the Pattern Finder smell usually indicate some form of inconsistency within the spreadsheet. This inconsistency can either be introduced due to an incomplete copy operation, or due to a mistake during the creation of the spreadsheet. Cells marked with the Pattern Finder smell may also contain temporary place-holder values, which were intended to be replaced eventually.

Consequences. Pattern Finder indicates inconsistencies within the bulk of a spreadsheet table. Cells within this area usually contain numeric input values or formulas which calculate interim results and are themselves referenced by other

formula cells either directly or by use of area-operations. Thus, inconsistencies within those areas usually lead to erroneous interim and final results.

Alleviation. As with Empty Cell, removal of the Pattern Finder smell is a straightforward process. We suggest to check whether the indicated cell holds the correct value type. Otherwise, determine which type and value should be contained and replace the faulty value. Suggestions for the missing content may be automatically derived from the surrounding context and presented to the user.

2.1.4 String Distance

Origin & Intent. The String Distance smell was first introduced in [Cunha *et al.*, 2012b]. Cunha *et al.* noticed, that typographical errors are frequently introduced during data input by typing. Consequently, they implemented the String Distance smell in order to detect typographical errors within spreadsheets. To that end, the smell indicates string cells which differ minimally from other strings contained within the same worksheet.

Target. String Distance is a spreadsheet smell which detects smelly string cells. As such, only cells containing string values may contain this smell. Other cell types, like formulas containing strings, are not taken into consideration during the detection of this smell.

Detection. In order to detect the String Distance smell, Cunha *et al.* proposed to utilize an algorithm introduced by Levenshtein in [1966]. This algorithm takes two strings as input and calculates the number of single transformation operations which need to be applied to one of the strings in order to transform it into the other. During the detection process, this algorithm is applied to each pair of strings within a row or column. If such a pair of strings only differs by one transformation, the string in question is signalled as smelly. Cunha *et al.* suggest to limit the detection to strings that contain more than three characters. Moreover, String Distance may indicate ascending numeric and alphanumeric designations within neighbouring cells as smelly, e.g. cells in a row or column that contain the values *Product 1*, *Product 2* et cetera. Sequences like this are common within spreadsheets and should therefore be excluded from the detection of this smell.

Example. The cell C9 of the *Sales* worksheet, displayed in Figure 2.1a, contains the String Distance smell. This cell contains the label "Productt two". By removal of a 't' character, this string can be transformed to the label contained in cells C6 to C8. Removal of a character only amounts to a single operation. Consequently, cell C9 is indicated as smelly.

Cause. Cells which contain the String Distance smell contain string values which differ minimally from other occurring strings. Such instances usually indicate faults which were introduced by typing errors during data entry. A cell may also be smelly of the String Distance smell as result of an accidental alteration of a cell value later on in the lifecycle of a spreadsheet.

Consequences. String cells are typically used as labels and headers within a spreadsheet. In such cases, relating faults may predominantly lessen the accountability of a spreadsheet. However, string values can also be used as part of conditional branches of a formula, or to encode values within the same row or column using a *LOOKUP* operator. According faults directly affect the corresponding calculations and lead to erroneous results.

Alleviation. Removal of the String Distance smell is a straightforward process. Simply check whether the string value contained in the indicated cell is correct. Otherwise, determine which string should be contained and insert it, replacing the faulty value. Suggestions as to which string should be contained may be automatically derived from the surrounding context and presented to the user, or even applied automatically.

2.1.5 Reference to Empty Cells

Origin & Intent. Reference to Empty Cells is a spreadsheet smell that was defined by Cunha *et al.* [2012b]. They pointed out that formulas which include references to empty cells are a typical source of errors in spreadsheets. Consequently, they introduced the Reference to Empty Cells smell to indicate such occurrences.

Target. The smell indicates formula cells which contain at least one reference to an empty cell. As such, only formula cells can contain this smell. Other cell types are not taken into consideration during the detection process of Reference to Empty Cells.

Detection. Detection of Reference to Empty Cells requires for every cell's formula within the spreadsheet to be analysed. For each formula, the cells that are referenced within the formula are determined. If one of those referred cells does not contain a value, the cell containing the formula in question is flagged as smelly with Reference to Empty Cells.

Example. Figure 2.1a depicts the *Sales* worksheet of our example. The formula at cell G6 of this worksheet contains a reference to the cell F6 which does not contain any value. As a result, G6 is marked as smelly.

Cause. Formula cells which are indicated to be smelly contain at least one reference that points to an empty cell. Such references may be introduced due to errors when entering formulas during spreadsheet creation. References to empty cells may also occur as a result of ill-considered copy & paste operations. When a formula cell containing a non-static reference is copied, the reference indices are updated according to the position difference between base and target cells of the copy operation. However, it is not guaranteed that the value type of the newly referenced cell complies with the requirements of the formula. Lastly, accidental alterations during the lifecycle of a spreadsheet may introduce references to empty cells into a formula or delete values from cells which are referenced by existing formulas.

Consequences. Formulas within spreadsheets are mainly utilized to conduct some form of calculation or statistical analysis based on provided input values. References to empty cells within a formula usually are not evaluated as errors. Instead, references to empty cells are interpreted as the numeric value 0, or an empty string, based on the related operator within the formula. However, formulas require meaningful input data at the location of their references to fulfil their intended function. Thus, a formula containing a reference to an empty cell may be syntactically valid, but will usually yield an erroneous result.

Alleviation. Removal of the Reference to Empty Cell smell requires analysis of the indicated formula as well as analysis of its references. In a first step, we suggest to check each reference as to whether it leads to an empty cell. If so, verify the correctness of the reference. If the reference is faulty, determine which cell should be pointed to instead and update the reference accordingly. If the reference itself is correct, examine the target cell in question. Determine which value or formula is missing and update the cell's content accordingly. Suggestions as to how the

reference could be corrected may be automatically derived from the surrounding context and presented to the user.

2.1.6 Quasi-Functional Dependencies

Origin & Intent. The spreadsheet smell Quasi-Functional Dependencies (QFD) was originally proposed by Cunha *et al.* in [2012b]. The idea for and the detection mechanism of this smell is based on the notion of Quasi-Functional Dependencies as described in [Abraham and Erwig, 2006a]. In general, a quasi-functional dependency is established by values of multiple rows or columns that are functional related to each other. The Quasi-Functional Dependencies smell indicates violations of such relations.

Target. QFD relies on the detection and cataloguing of functional dependencies between column or row values. Subsequently, only cells which contain values are relevant for this smell. Empty cells and cells which contain errors are not taken into consideration. Indeed, formula cells are ignored as well, as they do not contain constant cell values to form quasi-functional dependencies with.

Detection. Detection of the Quasi-Functional Dependencies smell requires the recognition of functional dependencies between at least two columns of a worksheet. Cunha *et al.* based their approach on [Chiang and Miller, 2008]. In this paper a more general version of the Functional Dependencies principle is presented and utilized to discern dirty values. Smell detection involves collecting and matching the entirety of all occurring spreadsheet values. Based on the results of the matching step, quasi-functional dependencies are synthesized. Lastly, existing quasi-functional dependencies are evaluated, and cell values that deviate from expected results are indicated as smelly.

Example. An occurrence of QFD can be found in cell F12 of the *Sales* worksheet, depicted in Figure 2.1a. Values within the columns A, B, C, and F follow a Quasi-Functional Dependency. The value of one cell within those columns can be inferred from specific other values within the other columns. For example, in rows 10, 11, and 13 the columns always contain the respective values 7, 1005237614, Product Three, and $19.99. However, cell F12 does not follow this established relation. It

contains the numeral value $18.99, while the values of the other columns infer the expected value of $19.99. This is suggestive of a typing error.

Cause. Cells which contain QFD usually indicate some form of inconsistency within the spreadsheet. A typing error or carelessness during data entry may be the cause of such faults. In addition, wrong or incomplete copy & paste may also lead to the introduction of values which smell of Quasi-Functional Dependencies.

Consequences. The Quasi-Functional Dependencies smell indicates an inconsistency within a row or column that contains input data of a spreadsheet. Such cells are usually referenced by formula cells either directly or by use of area-operations. As a result, inconsistencies indicated by QFD usually lead to erroneous interim and final calculation results.

Alleviation. Removal of a Quasi-Functional Dependencies smell is a straightforward process. Simply validate the value which is contained in the indicated cell. If the value is faulty, determine which value should have been contained and replace the faulty content. As only one specific value which alleviates the smell can be inferred, this refactoring may be provided as automated process.

2.1.7 Multiple Operations

Origin & Intent. Multiple Operations is a spreadsheet smell which was introduced by Hermans *et al.* in [2012b]. The smell is based on one of the most well-known code smells: the Long Method. Fowler proposed this code smell in [1999] as a way to indicate methods that feature an excessive number of statements. Such methods usually do not fulfil a single, specific purpose, but rather combine multiple tasks. However, the combination of tasks within a single method renders the method in question less comprehensible and should therefore be avoided. Similarly, formulas within a spreadsheet environment should feature a limited number of operations in order to facilitate their accountability. Therefore, Hermans *et al.* introduced the Multiple Operations smell to indicate formulas which feature an excessive number of operations.

Target. The spreadsheet smell Multiple Operations intends to find formula cells that contain an excessive number of operations. As such, only cells which contain

formulas are relevant for detecting this smell. Other cell types are not taken into consideration.

Detection. For the detection of the Multiple Operations smell, Hermans *et al.* suggest to count the number of operations each formula contains. Based on an evaluation of the EUSES spreadsheet corpus [Fisher and Rothermel, 2005], cells should be indicated as smelly as soon as a formula contains more than 4 operations.

Example. An occurrence of the Multiple Operations smell can be seen within cell D22 of the *Employees* worksheet, depicted in Figure 2.1b. The formula contained in this cell consists of 8 distinct operations. This number exceeds the suggested threshold for the detection of the Multiple Operations smell. Consequently, cell D22 is indicated as smelly.

Cause. Examination results of Hermans *et al.* suggest that the introduction of cells containing the Multiple Operations smell is an evolutionary process. Initially, formula cells do not contain excessive numbers of operations. However, spreadsheets are often target of recurring adaptations. Those adaptations lead to addition of operations to formulas as the need arises. Especially when working under time constraints, readability of a spreadsheet is often sacrificed in order to achieve a working solution.

Consequences. Cells that are flagged by the Multiple Operations smell contain a large number of operations. As the number of operations increases, the meaning of a formula becomes harder to understand. This circumstance is intensified by the fact that long formulas often are displayed cut-off, as they require more screen space than available.

Alleviation. In order to alleviate the Multiple Operations smell of a formula, we suggest refactoring: Split up the necessary calculations within the formula, distribute them over multiple cells and link them using references. As no additional knowledge or user-input is required, this refactoring may be provided as automated process. In some cases, multiple used operations may be replaced by a single area-supporting operation like **SUM**.

2.1.8 Multiple References

Origin & Intent. The spreadsheet smell Multiple References was first proposed in [Hermans *et al.*, 2012b]. Hermans *et al.* adapted the notion from a similar code smell: Multiple References. This code smell was presented by Fowler in [1999]. It indicates method definitions that feature an excessive number of parameters. Comprehensibility of a method definition declines as the number of parameters it requires increases. Likewise, the clarity of a formula declines as the number of references it features increases. Therefore, Hermans *et al.* suggested to utilize the Multiple References smell to indicate formula cells which require an excessive number of references.

Target. Multiple References points out formula cells which rely on a large number of different references. As a result, formula cells are exclusively relevant for the detection of this smell. Other cell types are not taken into consideration.

Detection. For detecting the Multiple References smell, Hermans *et al.* suggest to count the number of references to areas and other cells within a specific formula. Based on an evaluation of the EUSES spreadsheet corpus [Fisher and Rothermel, 2005], cells should be indicated as smelly as soon as they contain more than 3 references.

Example. Figure 2.1b depicts the *Employees* worksheet of our example. Cell D22 of this worksheet contains a formula referencing 6 unique areas and different cells. Consequently, it exceeds the suggested threshold for the detection of the Multiple References smell and is highlighted as smelly.

Cause. The introduction of Multiple References usually follows the same evolutionary process that leads to the introduction of the Multiple Operations smell. Formula cells normally start out requiring a limited number of references. However, during the lifecycle of the spreadsheet, adaptations are made to expand the functionality of those formulas which often require additional references.

Consequences. When contemplating spreadsheet formulas, the more single references to other cells they contain, the harder to comprehend they become. Similar to the Multiple Operations smell, this circumstance is intensified due to the fact that long formulas are usually not displayed completely, since they require more screen space than available.

Alleviation. In order to remove the Multiple References smell, we suggest to distribute the formula contained within the cell over multiple different cells, each containing a subset of the required operations and references. As no additional knowledge or user-input is required, this refactoring may be provided as automated process. In some cases, multiple used operations may be replaced by a single area-supporting operation like **SUM**. In such cases, the corresponding references can be united using area-references where appropriate. Additional references may be removed by relocating them next to and merging with an already referenced area.

2.1.9 Conditional Complexity

Origin & Intent. Conditional Complexity was introduced by Hermans *et al.* in [2012b]. The spreadsheet smell is based on a notion by Fowler regarding conditional operations in object-oriented code. According to Fowler, readability of code declines in instances where multiple nested conditional operators occur. Similarly, multiple nested conditional operators within a spreadsheet formula are difficult to comprehend. Therefore, Hermans *et al.* proposed the Conditional Complexity smell to indicate formulas featuring an excessive number of nested conditional operators.

Target. The smell Conditional Complexity detects formula cells that contain an excessive number of nested conditional operators. Consequently, the detection of this smell relies solely on cells which contain formulas. Other cell types are not taken into consideration.

Detection. For detecting the Conditional Complexity smell, Hermans *et al.* suggest to count the number of nested conditional operators contained in a specific formula. Based on an evaluation of the EUSES spreadsheet corpus [Fisher and Rothermel, 2005], cells should be indicated as smelly as soon as they contain at least 3 nested conditional operators.

Example. Occurrences of the Conditional Complexity smell are contained within cells D2, D9, and D15 of the *Employees* worksheet, illustrated in Figure 2.1b. Each of those formulas contains 3 nested conditional operations. Thus, each cell exceeds the suggested threshold for the detection of Conditional Complexity and is therefore marked to contain the smell.

Cause. According to the evaluation by Hermans *et al.*, this smell rarely occurs. Spreadsheet users obviously have some notion that conditional operators are complex. Subsequently, formulas containing conditional operators are usually handled with care. However, especially when working under pressure, users are less reserved. In such circumstances, users are more likely to rely on nested conditionals.

Consequences. Conditional Complexity marks formula cells within spreadsheets that contain multiple conditional operators. However, even a single conditional operator within a spreadsheet formula can be hard to comprehend for end users. This is owned to the fact that formulas are only afforded a limited amount of screen space to be displayed. This space usually does not suffice to show the entire conditional operator. Moreover, the syntax of conditional operators within spreadsheet environments does not emphasize the semantic function of each of its operands. This renders them hard to comprehend. Multiple consecutive conditional operators contained within a formula only worsen this effect.

Alleviation. One way to combat the Conditional Complexity smell is to divide the conditional operations over multiple cells. As no additional knowledge or user-input is required, this refactoring may be provided as automated process. Alternatively, the *SUMIF* and *COUNTIF* operators can be used to aggregate single conditional operators. If applicable, the LOOKUP operator can be used instead. This operator allows to specify a search key as well as a cell range containing condition-value-pairs. If the key matches one of the conditions within the cell range, the corresponding value is displayed.

2.1.10 Long Calculation Chain

Origin & Intent. The spreadsheet smell Long Calculation Chain was first proposed in [Hermans *et al.*, 2012b]. It can be seen as the antithesis to the Multiple Operations smell. In the usual workflow of a spreadsheet, formula cells rely on results of other formulas for their calculations. As a result, chains of dependent calculations are formed. However, in order to verify the correctness of such a chain, a spreadsheet user needs to trace along multiple references to find the origin of the input values. The longer such a chain grows, the harder it becomes to comprehend.

Therefore, Hermans *et al.* introduced this smell to indicate formula cells which rely on exceedingly long calculation chains.

Target. The spreadsheet smell Long Calculation Chain indicates formula cells that rely on a large number of successive dependent calculations. Consequently, only cells which contain formulas are relevant for the detection of this smell. Other cell types are not taken into consideration.

Detection. For detecting the Long Calculation Chain smell, Hermans *et al.* suggest to calculate the length of the longest path of successively referenced cells that need to be traced when evaluating a formula's value. Based on an evaluation of the EUSES spreadsheet corpus [Fisher and Rothermel, 2005], cells should be indicated as smelly as soon as the regarding metric is greater than 4.

Example. Cell C13 in the *Totals* worksheet, depicted in Figure 2.1c, is reliant on a vast number of cells. One of the longest calculation chains which can be formed is: Totals!C13 → Totals!C10 → Sales!H15 → Sales!G15 → Sales!G2 → Sales!E2. To calculate this chain, 5 dependencies need to be dereferenced. This number exceeds the threshold suggested by Hermans *et al.*. Thus, cell C13 of the worksheet *Totals* is indicated as smelly.

Cause. Formulas which rely on the interim results of other formulas for their calculations are a common practise within the spreadsheet domain. Consequently, the formation of calculation chains is a common occurrence. When expanding the functionality of a spreadsheet, users simply add new formulas that refer to existing calculation results. The overall structure of the utilized calculation chains is usually neglected during this process. Restructuring and regrouping of existing functionalities is not a common practice. Thus, long calculation chains are likely to occur, as the functionality of a spreadsheet expands.

Consequences. To understand the meaning and verify the correctness of a specific formula within a spreadsheet, a user needs to trace each reference within a given calculation chain. The longer a calculation chain grows, the higher the number of references and interim results it contains. Consequently, cells with long calculation chains are hard to comprehend and maintain.

Alleviation. In order to alleviate this smell while maintaining the functionality of the required calculations, we suggest to merge multiple steps along the calculation

chain into a single formula which aggregates all the necessary operations. As no additional knowledge or user-input is required, this refactoring may be provided as automated process. However, note that this approach leads to a trade-off between the intensity of this smell and the intensities of the Multiple Operations and Multiple References smells. In addition, if any cell within the calculation chain is referenced by another formula as well, this case has to be handled accordingly.

2.1.11 Duplicated Formulas

Origin & Intent. The Duplicated Formulas spreadsheet smell was introduced by Hermans *et al.* in [2012b]. This spreadsheet smell is based on the *Duplicated Code* smell presented by Fowler in [1999]. The related code smell indicates classes that contain multiple occurrences of similar code snippets. Likewise, different formula cells within a spreadsheet can contain equal formula-parts. The code smell Duplicated Formulas indicates such cells within a worksheet.

Target. Duplicated Formulas points out different formula cells that feature equal formula-parts. Consequently, the detection of Duplicated Formulas is reliant on formula cells only. Other cell types are not taken into consideration.

Detection. For detecting the Long Duplicated Formula smell, Hermans *et al.* suggest to utilize the *relative R1C1 notation*. References depicted in this notation express their references to other cells relative to the cell which contains the formula. For example, the formula *MAX(A1:A3)* in cell A4 would be written as *MAX(R[-3]C:R[-1]C)* in this notation. Hermans *et al.* suggest to measure the number of cells within a worksheet that feature formula-parts being either equal or share the same *relative R1C1 notation*. Based on an evaluation of the EUSES spreadsheet corpus [Fisher and Rothermel, 2005], cells should be indicated as smelly as soon as they share the same formula-part with at least 6 other cells. Nevertheless, in [2012b] they present an example whereby the Duplicated Formula smell is indicated for a cell which shares its sub-formula with merely a single other cell, casting a doubt on the provided threshold. Lastly, formulas which are entirely equal in *relative R1C1 notation* are not marked as containing the smell, as this is common practice among spreadsheet users.
Abreu *et al.* proposed an adapted approach for the detection of this smell in [2014a].

Instead of relying on the *relative R1C1 notation*, they directly compare occurring formulas. As an example, two cells containing the formulas SUM(A1:A3) * 1.1 and SUM(A1:A3) * 1.2 would have the first part duplicated and therefore be indicated as smelly.

Example. An occurrence of the Duplicated Formulas smell is demonstrated by cell H13 within the *Sales* worksheet, depicted by Figure 2.1a. The part of the formula referencing the cell values to the left of the formula is shared with other formulas within column H. However, the constant multiplication factor 0.9 of the formula within H13 differs from the remaining column. Thus, the cell is indicated to be smelly. Another example can be seen in cell H14 of the same worksheet. The formula of this cell shares the *SUMIF* part with its neighbour, but expands it by an additional multiplication. Consequently, it is pointed out as containing the Duplicated Formulas smell.

Cause. Duplication and adaptation of formulas is common practice during the creation of a worksheet. It is not surprising that multiple occurrences of similar or equal formula-parts are contained in various formula cells within the same worksheet. Evaluation by Hermans *et al.* indicates that a substantial amount of spreadsheet users understand that formula duplication can lead to problems. However, they also observe that comprehension of the issue by users does not lead to a lesser degree of occurring duplications within the spreadsheets of those users. Moreover, some users do not see any harm at all in duplicating formulas.

Consequences. Duplication of formula parts may lead to a number of problems. If a large part of a given formula is duplicated from another one, it is hard to distinguish them within the program interface. Moreover, if a formula containing duplicated parts needs to be adapted, this adaptation has to be applied to each of the duplicated formulas. This poses a threat to the maintainability of the spreadsheet, as it is easy to overlook one of the duplicated formulas by mistake.

Alleviation. To remove the Duplicated Formulas smell, we suggest to extract the duplicated section from each related formula and move it into a single, designated formula cell. The base cells need to be updated with the reference to the new formula cell accordingly. This procedure usually increases readability and maintainability of a worksheet. As no additional knowledge or user-input is required, this refactoring may be provided as automated process.

2.1.12 Inappropriate Intimacy

Origin & Intent. Inappropriate Intimacy is an inter-worksheet smell which was proposed by Hermans *et al.*. According to their definition in [2012a], the smell is primed to detect worksheets which are too heavily reliant on the content of other worksheets. The smell is based on the identically named code smell presented by Fowler in [1999].

Target. Inappropriate Intimacy intents to detect worksheets that feature an excessive number of references to different worksheets. Such references are only contained within formula cells of a spreadsheet. As a result, the detection of the Inappropriate Intimacy smell is solely reliant on analysis of formula cells. Other cell types are not taken into consideration.

Detection. In order to detect an inappropriately intimate worksheet, Hermans *et al.* introduced the so-called *Intimacy* metric. *Intimacy* between two spreadsheets is measured by the number of connections between them. The function, defined by this metric has the type $\mathbf{W} \times \mathbf{W} \rightarrow int$ and is defined by the formula:

$$\text{Intimacy}(w_0, w_1) \equiv |\{(c_1, c_0) \in \mathbf{KS} : c_0 \in w_0 \wedge c_1 \in w_1 \wedge w_0 \neq w_1\}|$$

Intimacy counts the number of pairs (c_1, c_0) in $\mathbf{KS}$, the set containing all connections of the spreadsheet, for which holds true that c_0 is contained in worksheet w_0, c_1 is contained in worksheet w_1, and the two worksheets are different. Thus, it counts the number of references the worksheet w_0 contains which point to cells in the worksheet w_1. According to this definition, multiple references from one worksheet to a specific cell of another worksheet are counted repeatedly. The overall intimacy of a worksheet is calculated as follows:

$$\text{II}(w_0) \equiv \max\{\text{Intimacy}(w_0, w_1) : w_0, w_1 \in \mathbf{S}\}$$

This metric indicates the maximum intimacy the worksheet w_0 has with any other worksheet. Hermans *et al.* suggest to utilize this metric to detect the Inappropriate Intimacy smell. Based on an evaluation of the EUSES spreadsheet corpus [Fisher and Rothermel, 2005], the threshold for the detection of this smell is 8. Thus, a

worksheet should be highlighted as smelly as soon as it contains at least 8 references to another worksheet.

Example. The detection of the Inappropriate Intimacy smell is based on the maximal number of references to a different spreadsheet. Figure 2.1c depicts the *Totals* worksheet of our example, which contains this smell. By analysing formulas occurring in this worksheet, we can see that it contains 6 references to the worksheet *Employees* and 9 references to the worksheet *Sales*. Therefore, the maximal number of references to a different worksheet for the *Totals* worksheet is 9. The value exceeds the suggested threshold for this metric. As a result, the *Totals* worksheet is indicated as smelly.

Cause. According to the evaluation by Hermans *et al.* in [2012a], the Inappropriate Intimacy smell is quite common among typical spreadsheets. They observed that spreadsheet creators are usually not trained as programmers. Therefore, structuring spreadsheets in a logical way poses a difficult task. Those novice users consequently rely excessively on cross-worksheet references within their formulas. This leads to the accumulation of the Inappropriate Intimacy smell in some cases. In particular, Hermans *et al.* identified two cases which repeatedly cause the Inappropriate Intimacy smell. One depicts the use of a so-called auxiliary worksheet in combination with a second one: The auxilary worksheet contains data on which the other worksheet relies. The second case depicts two worksheets which repeatedly reference each other without any clear distinction of purpose between them.

Consequences. A worksheet containing the Inappropriate Intimacy smell indicates the existence of a strong semantic connection to at least one other worksheet. This split of functionality between multiple worksheets may weaken understandability of the spreadsheet as a whole. Moreover, when changes are made within the referenced worksheet, the reliant worksheet needs to be checked for correctness as well. This requires a spreadsheet user to continuously switch between the two worksheets.

Alleviation. Removal of the Inappropriate Intimacy smell of a worksheet requires to reduce its number of references to at least one specific other worksheet. However, this reduction usually requires a well thought-out restructuring process, as the functionality of the spreadsheet as a whole needs to remain unaltered. The correct refactoring strategy to apply depends on the individual situation. Wherever

possible, neighbouring references to other worksheets should be merged using area references. In some cases, importing more extensive parts of functionality from the referenced worksheet may be required. In other cases, the best course of action is to merge two worksheets.

2.1.13 Feature Envy

Origin & Intent. Feature Envy is an inter-worksheet smell proposed in [Hermans *et al.*, 2012a]. Hermans *et al.* derived it from a corresponding smell for object-oriented code. Fowler introduced this code smell in [1999] to indicate that a specific method is more reliant on the fields of another class than on fields of the class which contains the method. The same principle can be applied to spreadsheet formulas. The spreadsheet smell Feature Envy indicates formulas which are excessively reliant on foreign worksheets.

Target. Feature Envy detects and indicates formula cells which feature an excessive number of references to foreign worksheets. Therefore, only formula cells are analysed during the detection process of this smell. Other cell types are not taken into consideration.

Detection. In order to detect feature envious formula cells, Hermans *et al.* introduced the so-called *Enviousness* metric. *Enviousness* indicates how many references to cells of other worksheets a formula contains. The metric defines a function of the type $\mathbf{C} \rightarrow int$ and is defined by the formula:

$$\mathrm{FE}(c_0) \equiv |\{(c_1, c_0) \in \mathbf{KS} \mid \exists w : c_0 \in w \wedge c_1 \notin w\}|$$

This metric counts the number of pairs (c_1, c_0) in **KS** for which holds true that c_0 is contained in the worksheet but the cell c_1 is not. Hermans *et al.* suggest to utilize this metric to detect the Feature Envy smell. Based on an evaluation of the EUSES spreadsheet corpus [Fisher and Rothermel, 2005], a formula cell should be indicated as smelly as soon as it contains at least 3 relations to other worksheets.

Example. An example of the Feature Envy smell can be found in cell C14 of the *Totals* worksheet, depicted in Figure 2.1c. By analysing the formula in cell C14, we can see that it contains 4 distinct references to other worksheets. This value exceeds

the threshold which is suggested for the detection of the Feature Envy smell. Thus, cell C14 of the worksheet *Totals* is indicated as smelly.

Cause. The introduction of this smell is based on the same principles which apply to the Inappropriate Intimacy smell. Hermans *et al.* state that Feature Envy occurs quite often among spreadsheets [2012a]. They reason that spreadsheet creators are usually not trained as programmers. Structuring spreadsheets in a logical way is not a trivial problem. As a result, such users rely heavily on cross-worksheet references within their formulas, resulting in the accumulation of the smell. Hermans *et al.* pointed out two distinct cases which repeatedly lead to the introduction of Feature Envy: One depicts the use of so-called auxiliary worksheets which contain data in combination with other worksheets that rely on the provided data. The second case depicts two worksheets which repeatedly reference each other without any clear distinction from each other.

Consequences. Feature Envy implies, that a specific cell is excessively interested in cells from another worksheet, rendering it harder to understand. This is due to the fact, that spreadsheet users heavily rely on the highlighting feature that indicates value ranges relevant to a formula. However, in most common spreadsheet programs this feature does not work across worksheets.

Alleviation. To alleviate the Feature Envy smell, we suggest to move the formula in question to the corresponding worksheet and link the result of the computation back to the initial worksheet. By carrying out this procedure, the formula in question is moved closer to the cell it is referring to. As a result, the comprehensibility of the worksheets should be improved.

2.1.14 Middle Man

Origin & Intent. Middle Man was introduced as an inter-worksheet smell in [Hermans *et al.*, 2012a]. Hermans *et al.* defined a middle man in the context of spreadsheets as formula which only contains a single reference to another cell. Such formulas are used frequently as sources of information to be used in further calculations within a local worksheet. However, worksheets which contain an excessive number of such middleman cells are deemed smelly. The notion of a middle man was derived from an identically named code smell introduced by Fowler [1999].

Target. The Middle Man smell indicates worksheets that contain an excessive number of middleman cells. A middleman cell is defined as a formula that merely features a single reference. Thus, the detection of this smell is solely reliant on the analysis of formula cells within a specific worksheet. Other cell types are not taken into consideration.

Detection. In order to detect a worksheet which is smelly of the Middle Man smell, Hermans *et al.* introduced a special type of formula: the *middle man* formula. A middle man formula fulfils the sole purpose of fetching a value from another cell. Cells which contain such a formula are detected by the function MMF: $\mathbf{C} \rightarrow bool$. However, for the Middle Man smell to be detected for a worksheet, a calculation chain of two consecutive passing formulas needs to occur. To that end, Hermans *et al.* suggest the following metric:

$$\mathrm{MM}(w) \equiv |\{(c_1, c_0) \in \mathbf{KS} : c_1 \in w \wedge MMF(c_0) \wedge MMF(c_1)\}|$$

This metric counts the numbers of middle man formulas in a worksheet that are themselves used as reference by another middle man formula. Hermans *et al.* suggest to utilize this metric to detect the Middle Man smell. Based on an evaluation of the EUSES spreadsheet corpus [Fisher and Rothermel, 2005], a worksheet should be indicated as smelly as soon as it contains at least 7 chained middle man formulas.

Example. An example related to the Middle Man smell can be found in cell B21 of the worksheet *Employees* (see Figure 2.1b). Analysis of the formula in cell C14 indicates, that this cell is a middle man cell. Its sole purpose is to fetch the value contained in cell C1 of the *Totals* worksheet. However, the cell B21 itself is target of a middle man cell as well: cell B15 within the worksheet *Sales*. Thus, a chain of multiple middle man cells is established. This, in turn, increases the suspiciousness of the *Employees* worksheet towards the Middle Man smell. However, the suggested value of at least 7 linked middle man cells is not reached. Therefore, the worksheet is not indicated to contain the Middle Man smell.

Cause. Middle man cells are commonly introduced to pass along values and calculation results throughout different worksheets. The resulting values are often used for further calculations. However, middle man cells may also be utilized to keep an eye on specific values in different parts throughout a spreadsheet. In some

cases, values are even passed along within the same spreadsheet for that effect. In either case, the method of value passing can follow one of two different approaches. Using the first approach, values are passed from the source to each middle man individually. Using the second approach, values are passed along by formation of a chain. Although, this approach may introduce a number of problems.

Consequences. Middle man cells are commonly used to state results of calculations within other worksheets. However, a problem arises when a worksheet contains an excessive number of such middle man cells. In this case, the worksheet in question may lack enough logic to justify being a separate worksheet. Such occurrences impair general readability of a spreadsheet.

Alleviation. Which strategies should be applied to alleviate the Middle Man smell depends on the circumstances in which the middle man cells are utilized. In cases where values are passed along a chain of multiple middle man cells, we suggest to adapt the corresponding formulas of each step along the chain to directly reference the source cell instead. As no additional knowledge or user-input is required, this refactoring may be provided as automated process. In cases where middle man cells relay values for further processing to different worksheets it might be favourable to simply relocate the spreadsheet logic in question to the vicinity of the data source, thus removing the requirement for value passing altogether.

2.1.15 Shotgun Surgery

Origin & Intent. Shotgun surgery is an inter-worksheet smell presented by Hermans *et al.* [2012a]. This smell is based on the corresponding code smell which indicates that a change within one class needs to be followed up by numerous little changes in several reliant classes. This code smell was presented by Fowler in [1999]. The same principle can be applied to spreadsheets. Thus, the Shotgun Surgery smell is detected when a specific cell is referenced by multiple formulas within different worksheets.

Target. The Shotgun Surgery smell indicates worksheets which are referenced by a large number of formulas which are contained in different worksheets. Therefore, the only formula cells are required for the detection of this smell. Other cell types are not taken into consideration.

Detection. In order to detect worksheets which contain the Shotgun Surgery smell, Hermans *et al.* introduce two metrics, both of the type $\mathbf{W} \rightarrow int$:

$$\text{ChangingFormulas}(w) \equiv |\{(c_1, c_0) \in \mathbf{KS} : c_1 \in w \wedge c_0 \notin w\}|$$

The metric Changing Formulas counts the number of formulas which refer to a cell in worksheet w. Based on an evaluation of the EUSES spreadsheet corpus [Fisher and Rothermel, 2005], a worksheet should be deemed smelly as soon as it is referenced by at least 9 changing formulas.

$$\text{ChangingWorksheets}(w_0) \equiv |\{w_1 \in \mathbf{S} \mid \exists (c_1, c_0) \in \mathbf{KS} : c_0 \in w_1 \wedge c_1 \in w_0\}|$$

The metric Changing Worksheets counts the number of worksheets which contain references to any cell in the worksheet w_0. Based on the evaluation of the EUSES spreadsheet corpus, a worksheet should be indicated as smelly as soon as it is referenced by at least 2 other worksheets.

Example. The *Employees* worksheet contains the Shotgun Surgery smell. As can be seen in Figures 2.1a and 2.1c, cells within this worksheet are referenced by both the *Sales* and the *Totals* worksheet. Therefore, the threshold to indicate the Shotgun Surgery smell for this worksheet is breached.

Cause. The Shotgun Surgery smell within a spreadsheet can usually be accounted to a lack of foresight during set-up of a spreadsheet. When working with extensive spreadsheets, references to cells of foreign worksheets are often used. Limiting the amount of such references requires a well-considered spreadsheet structure. However, most spreadsheets are created by novice users, rather than professional programmers. For those users the introduction of clean and comprehensible structures is usually a minor concern. Consequently, instances of Shotgun Surgery naturally accumulate as the functionality of novice-spreadsheets expands.

Consequences. The inter-spreadsheet smell Shotgun Surgery indicates the occurrence of a cell which is referred to by many different formulas throughout other worksheets. Naturally, if the target cell changes, many of the formulas which refer to it need to be adapted as well. Thus, Shotgun Surgery poses an influence on the maintainability of a spreadsheet. In addition, the introduction of faults into

the spreadsheet is more likely, as a user might forget to adapt one of the reliant formulas.

Alleviation. In order to cope with the Shotgun Surgery smell, we suggest to minimize the references to cells within the worksheet in question where possible. Another approach to alleviate this smell is to import the functionality of formulas which rely on cells within the worksheet into the worksheet in question.

2.2 Overview and Comparison

Spreadsheet smells can be arranged into groups based on which properties of spreadsheets they are related to. As an example, multiple smells are concerned with the complexity and accountability of formula cells. Thus, Long Calculation Chain, Multiple Operations, Multiple References are connected to one another. When Long Calculation Chain is lowered, others are increased and the other way around. Those trade-offs exist in source code smells, too.

Empty Cell detection is a subset of the methodology to detect the Pattern Finder smell. Albeit Empty Cell detection usually allows for more general thresholds as to which part of the local neighbourhood is analysed.

Multiple occurrences of Feature Envy in relation to a specific worksheet imply Inappropriate Intimacy, but not the other way around. Middle Man may be seen as extreme form of Inappropriate Intimacy, of referring to a single worksheet. Middle Man is also an extreme case of Feature Envy, as corresponding formulas only refer to external cells. Shotgun Surgery may be caused by feature-envious or Middle Man cells, but is not required to be so. Shotgun Surgery can also be seen as antithesis to the Inappropriate Intimacy smell. Whereas Inappropriate Intimacy indicates that a given worksheet is too reliant on a number of different worksheets, Shotgun Surgery indicates that other worksheets are too reliant on a specific cell within a single worksheet.

Every proposed inter-worksheet smell can be removed by simply placing all required spreadsheet logic into a single worksheet. However, in general a more structured design-approach improves readability and maintainability of spreadsheets. Migration of spreadsheet logic into a single worksheet in reality is only advisable up to

a specific point. This property is characteristic for an optimization process. Nevertheless, none of the proposed smells is adequate to be used as counterbalance for such a process. This indicates a vacancy within the current spreadsheet smell catalogue.

In general, smells which indicate similar problems or have similar detection methods tend to influence one another. In order to highlight fundamental similarities and differences Table 2.1 provides an overview of the spreadsheet smells presented in this paper. Within the table, the following properties are evaluated:

Name	The name of the spreadsheet smell.
Target	Which parts of a spreadsheet can contain the smell.
Oo-pendant	Which code smell this smell is based on, if any.
Cause	At which stage of the spreadsheet lifecycle the smell is usually introduced.
Consequences	Which aspects of spreadsheet quality are affected by the smell.
Alleviation	Whether the smell can be removed automatically, assisted or manually.

Name	Target					Oo-pendant	Introduction			Consequences			Alleviation		
	Empty cells	Numeric cells	String cells	Formula cells	Worksheets		Creation	Data entry	Expansion	Erroneous result	Impeded quality		Manual	Assisted	Automated
Std. Deviation		•				-		•		•			•		
Empty Cell	•					-	•	•		•				•	
Pattern Finder	•	•	•	•		-	•	•		•				•	
String Distance			•			-		•		•					•
Ref. to Empty Cells				•		-	•		•	•				•	
QFD		•	•			-		•		•					•
Multiple Operations				•		Long Method			•		•				•
Multiple References				•		Many Parameters			•		•				•
Cond. Complexity				•		-	•		•		•				•
Long Calc. Chain				•		-	•		•		•				•
Duplicated Formulas				•		Duplication			•		•				•
Inappr. Intimacy					•	Inappr. Intimacy	•		•		•		•		
Feature Envy				•		Feature Envy	•		•		•		•		
Middle Man					•	Middle Man			•		•		•		•
Shotgun Surgery					•	Shotgun Surgery	•		•		•		•		

Table 2.1: Comparison of Spreadsheet Smells

2.3 Utilization of Spreadsheet Smells

In the previous sections, we summarized which spreadsheet smells have been proposed by the community and how those smells can be detected. However, in order to benefit from smell detection, information about perceived smells needs to be applied in some form. Throughout the various examinations of the topic within the scientific community, a number of different approaches have been devised to that end, which we organize into the following 3 categories:

- *Smell Indication* approaches directly provide feedback about detected smells to the user. Feedback may be provided either in-place or in form of additional documents. Users may then manually adapt their spreadsheets based on the additional information.
- *Smell Removal* approaches go a step further and attempt to process ways to remove perceived smells from a spreadsheet. Removal of smells usually requires some form of refactoring of the spreadsheet structure. Inferred changes which lead to the removal of smells may either be applied automatically or presented to the user in form of change suggestions.
- *Other Approaches* consult information about perceived smells and conduct further processing to attain a goal which is relevant for the approach in question. For example, Abreu *et al.* proposed to use smells as input for a fault localization algorithm [2014a]. This algorithm consequently determines a set of cells which is not necessarily smelly in itself but is likely to cause the smells contained in the spreadsheet.

In the following section, we describe each category in more detail and state exemplary existing approaches for each of them.

2.3.1 Smell Indication

The straightforward approach to utilize the results of smell detection is to accumulate and provide feedback about perceived smells within the spreadsheet. Feedback may be provided either in-place via visual cues or in form of additional documents,

diagrams, and charts. Most initial approaches towards the study of spreadsheet smells used such feedback mechanisms. The focus of those early works has been to establish and refine valid smells and detection processes. Smell utilization has been a minor concern at that stage. However, user benefit may still be gained by those approaches. If applied, users may inspect, re-evaluate, and update their spreadsheets based on the additional feedback-information.

Hermans *et al.* were among the first to venture into the scientific field of spreadsheet smells in [2012a]. Their work revolves around detecting and visualizing inter-worksheet smells within spreadsheets. In a previous work [Hermans *et al.*, 2011], they already established a process to extract data flow diagrams from spreadsheets as a suitable option to visualize the inherent structures of spreadsheets. Such diagrams usually provide visual aid to comprehend data dependencies and relationships between processes in information systems. Hermans *et al.* extended their computed diagrams to also indicate detected inter-worksheet smells. Following the paradigm established in [2011], boxes which represent worksheets are highlighted using color if they contain a smell. The hue of the color indicates the intensity of the detected smell. In addition, tool-tips explain which smell was detected and the concrete location of smell-inducing cells within the respective worksheet.

Another approach by Hermans *et al.* relying on smell indication is presented in [2012b]. In this work, Hermans *et al.* focus on the application of existing code smell principles to formula cells within spreadsheet environments. In contrast to their previous work [2012a], inter-worksheet dependencies are not considered. As a consequence, the authors chose a different visusalization method for detected smells as well. Taking cues from related work like [Abraham and Erwig, 2007], they adapted the notion of risk maps to indicate smells in specific formula cells. A 3-tiered, color-coded overlay over the spreadsheet is used to highlight affected cells. The more intense the color, the more likely a cell contains a smell. In addition, comments are added to each coloured cell, providing an explanation about the suspected smell.

Cunha *et al.* proposed another example which features the informative approach to smell utilization. They introduced the tool *SmellSheet Detective* [2012b] [2012a]. The purpose of this tool is to detect and indicate a predefined set of smells within provided spreadsheets. Both, spreadsheets stored within the Google Docs platform as well as locally stored spreadsheets, may be analysed by *SmellSheet Detective*. The

tool itself is based on a modular and extensible Java library, which allows for easy incorporation of new smells into the detection process. The result of this process may be exported in form of either *csv*, *Excel*, or LaTeX tables.

Lastly, Hermans *et al.* [2013] presented another approach in the field of smell detection. In this work, they attempt to automatically detect and highlight data clones in a provided spreadsheet. Data clones are the result of copy-paste operations within a spreadsheet. Not only single cells, but also groups of cells which are likely to be copied are indicated by their approach. Visualization is conducted based on a combination of the techniques, conducted in the authors' previous approaches [2012a] [2012b]. A data flow diagram is created which indicates data clone dependencies between worksheets via directed arrows. In addition, comments are added to affected cells and areas which explain either where specific values were copied to or where the source of specific values can be found within the spreadsheet.

2.3.2 Smell Removal

Although perceived smells do not always indicate errors, they at least point out some flaw which usually can be improved in some form. However, many spreadsheet smells are based on structural properties. Consequently, to remove such smells changes to the structure of the spreadsheet are required. Such changes require substantial effort and may lead to the introduction of new issues during the process. Consequently, spreadsheet users often shy away from manually fixing those flaws, even if indicated by a tool as risky. Approaches which fall into the removal category of smell utilization attempt to support users in such circumstances by automatically inferring the changes required to remove a specific smell. However, in some instances more than one course of action can be established to remove a perceived smell while none of the possibilities can be procedurally determined as preferred fix for the circumstance. In such cases, a list of possible fixes and/or additional information is generated and proposed to the user. The user may then choose which action to take.

Badame and Dig proposed an approach to automated smell removal in [2012]. Their tool, dubbed *REFBOOK*, provides spreadsheet users with access to a suite of automatic refactorings, each removing one commonly encountered spreadsheet

smell. The tool is available in form of a plug-in for Microsoft Excel. However, due to the multi-tiered architecture of their tool, it can be easily adapted for other spreadsheet environments as well. Refactoring options are provided to the user via a custom entry in the context menu. After selecting the target cell range of the operation and activating the context menu, a user simply has to choose the desired refactoring from the provided list. The plug-in handles communication of the selected command to the back-end of the tool as well as application of the necessary changes to the spreadsheet. The back-end process is working based on a system of generic spreadsheet-entities. This allows the authors of the tool to expand it to other spreadsheet programs by simply supplying a matching add-on for the desired platform while the back-end does not need to be altered. The authors' evaluation of the tool indicates that users in general prefer the refactored output over the initial spreadsheet. Usage of the *REFBOOK* plug-in resulted in average time-savings of more than 50 % in comparison to manual refactoring based on the same conditions. Moreover, manual refactoring frequently introduces new faults into the spreadsheet, whereas refactorings based on *REFBOOK* does not.

After their extensive work on smell detection, Hermans *et al.* proposed an approach which incorporates removal of detected smells as well [2014]. This approach is based on their previous work regarding smells affecting spreadsheet formulas, presented in [2012b]. Rather than just visualizing detected smells, Hermans *et al.* also attempt to infer refactoring processes to remove those smells from the spreadsheet. Concrete refactoring suggestions are consequently added to the comments of each affected cell. Evaluation of their approach suggested, that some detected smells can be reliably removed using the inferred refactoring suggestions. However, spreadsheet users might struggle to implement those refactorings themselves.

2.3.3 Other Approaches

The last category regards approaches which introduce other ideas to utilize the results of smell detection. Smell detection itself is not the focus of such approaches. Rather, information about detected smells is used as an interim result to base further processing on. For example, smell information may be combined with other static analysis techniques to improve accuracy or coverage ratings of existing spreadsheet

fault location processes. Not much work has been published in this regard as of yet. However, we suspect that more advances will make use of hybrid approaches involving spreadsheet smells in the future.

Abreu *et al.* proposed an approach which revolves around further processing of smell detection results [2014a]. They presume that even if only one cell is indicated as smelly, a number of other cells are likely to contribute to the suspicious circumstance as well. Following this notion, a subset of cells which was identified as smelly beforehand is provided as input for a fault localization algorithm. This algorithm determines a set of cells which is not necessarily smelly in itself but is likely to cause the smells contained in the spreadsheet. Abreu *et al.* implemented this process in a tool dubbed *FaultySheet Detective* [2014b]. It represents an extension of the *SmellSheet Detective* tool, proposed in [Cunha *et al.*, 2012a]. In addition to the expanded detection process, *FaultySheet Detective* supports a larger set of spreadsheet smells as basis of analysis. The tool supports analysis of spreadsheets stored within the Google Docs platform as well as locally stored spreadsheets. Suspicious cells within the spreadsheet provided as result by the tool are indicated via background color. The intensity of the color hue correlates with the number of faults which were found within the respective cell. In addition, a note is added to each coloured cell explaining which smells are detected for the cell as well as stating the result of the fault localization algorithm for the cell. Evaluation which was based on a faulty spreadsheet catalogue indicates that *FaultySheet Detective* is capable of identifying more than 70 % of faults within the tested spreadsheets.

3. Label-Based Reasoning about Units and Dimensions

Static checking has long since been an integral part of software technology. However, the application of those principles to the domain of spreadsheets is a more recent development. The notion of unit-checking of spreadsheets, in particular, was introduced by Erwig and Burnett in the early 2000s. In [2002] they proposed a formal unit system for spreadsheets which does not rely on the concept of types but rather uses a concrete notion of units. Units thereby are inferred from information already existing in the spreadsheets. Based on this unit information, the system allows to reason about and check the integrity of formulas as soon as they are typed in.

To provide an example, suppose the user has created the spreadsheet depicted in Figure 3.1. The labels placed by the user imply that column B is related to Espresso. Based on the formulas, we can deduce that the total in Cell B5 is also related to the unit Espresso. In regard to the provided formula, this is a valid combination of units. Cell D3 adds the units Espresso and Decaf. This may seem illegal initially. However, the added cells, B3 and C3, also refer to a second unit: July. Thus, D3 combines the units (July, Espresso) and (July, Decaf). The result can be reduced to the unit (July, Coffee), which is a legal combination. However, suppose the user changes the formula in D3 to now calculate $B2 + C3$ instead. The formula would now compute the sum of (July, Espresso) and (August, Decaf). In this instance, both unit-parts mismatch: Espresso vs. Decaf and July vs. August. Consequently, a unit-checking system would indicate an error.

Current approaches towards unit-checking of spreadsheets still follow the basic principle presented in the example above. Nevertheless, various authors proposed a number of refinements and variations based on this principle. In the following

	A	B	C	D
1		Coffee		
2	Month	Espresso	Decaf	Total
3	July	31	22	53
4	August	42	23	65
5	Total	73	45	118

(a) Value view.

	A	B	C	D
1		Coffee		
2	Month	Espresso	Decaf	Total
3	July	31	22	=B3+C3
4	August	42	23	=B4+C4
5	Total	=B3+B4	=C3+C4	=D3+D4

(b) Formula view.

Figure 3.1: Example spreadsheet depicting coffee consumption.

subsections, we state an overview of the most important approaches. We proceed by highlighting some of the basic principles that most of these approaches rely on. Lastly, we point out some derivative concepts which were based on those ideas.

3.1 History

Unit-checking within spreadsheets is based on the principle of dimensional analysis within programming languages. The method was first adapted to the domain of spreadsheets in the early 2000s. At that time, type-checking was already an established static code analysis practice within the scientific community. However, application of static analysis to the domain of spreadsheets was a novel approach. Two reasons can be stated for this circumstance: Firstly, the procedures and results of static type checking approaches are hard to grasp for users. Indeed, users of spreadsheet programs usually are not trained programmers. Consequently, those users are often thought to be unable or unwilling to handle the extra effort associated with static checking approaches. Secondly, spreadsheet programs in general were not taken seriously within the programming languages community at the time. Nevertheless, the number of end-user programmers relying on spreadsheet programs kept on increasing. In addition, researchers proposed extensive evidence indicating that

20 % to 40 % of the spreadsheets produced and maintained by novice programmers contain errors, leading to significant economical losses.

Erwig and Burnett recognized this need for improvement. In 2002, they proposed a unit system for spreadsheet programs based on general research into units and dimensions [2002]. Following their definitions, the notion of a "unit" should be application dependent, rather than adapting the meaning of units to represent scales of measurement for specific dimensions. Their proposed system attempts to detect illegal combinations of units within spreadsheet calculations. To that end, the system should support manual annotation of cells with explicit unit information. However, the unit system should also infer as much information as possible without requiring additional user interaction. This requirement is indicative of a "gentle slope" language feature, as described in [Myers *et al.*, 1992]. Throughout their work, Erwig and Burnett formulate a formal spreadsheet calculus and define their notion of headers and units within a spreadsheet environment. Based on these definitions, they propose the first formal set of unit-inference rules for spreadsheets in scientific literature. In addition, they state starting points for header inference as well. In [2002], Erwig and Burnett expand their work and describe a visual system which supports the formal definitions stated in [Erwig and Burnett, 2002]. The proposed system provides two main features: Firstly, it allows for visual explanation of unit-inference mechanics to end-users. Secondly, it aids these users in customizing the system's inference rules.

Inspired by the work of Erwig and Burnett, Ahmed *et al.* proposed a rule-set for unit-checking of spreadsheets of their own [2003]. The unit inference rules proposed by Ahmed *et al.* are simpler than the rule-set proposed by Erwig and Burnett. However, Ahmed *et al.*'s system adds the notion of semantic relationships between headers to the process and supports a wider range of operators. According to their definition, two headers may either entertain an *is-a* relationship, signalling some kind of generalization between the headers, or a *has-a* relationship, indicating that one header describes a property or item of the related header. Ahmed *et al.* implemented their unit-checker based on a common programming environment. The resulted tool allows for unit-checking of spreadsheets within Microsoft Excel. However, as no header-inference algorithm was implemented, each cell needs to be manually annotated with its unit information. Evaluation conducted by the authors shows

that the tool is able to detect errors in many cases. Coverage and false-positive rates are unknown.

Antoniu *et al.* proposed a tool for validating dimension correctness of spreadsheets dubbed *XeLda* [2004]. The tool supports analysis of spreadsheets based on *Excel*. Unlike previous approaches, Antoniu *et al.* regard units as dimensions associated with a numeric value. Based on this definition, a unit consists of a possibly empty list of unit-exponent pairs (e.g. $m^1 s^{-1}$). Unit inference follows the mathematical principle of unit combination based on employed operators. For example, a formula calculating the multiplication of the units $m^1 s^{-1}$ and s^1 results in the unit m^1. Detected unit conflicts may be resolved via user-defined unit coercions. Initial unit information is required for the inference process. This information can be supplied to the system via manual annotation of numeric input and formula cells. The unit inference process is able to discern two distinct error types: *Match errors* indicate that a derived unit deviates from a formula's annotated expectation. *Consistency errors* indicate that a calculation relies on inconsistent units. Erroneous cells are highlighted via background color and a descriptive comment. In addition, the tool displays arrows pointing from the cells which contribute to the error via references to the erroneous cell itself. Initial testing of *XeLda* by Antoniu *et al.* indicates that the tool is able to accurately detect dimension errors within annotated spreadsheets.

In 2004, Abraham end Erwig proposed their approach to conquer the yet unsolved problem of automated header inference [2004]. Throughout their work, they propose a collection of algorithms to infer header information from a spreadsheet. Each of these algorithms is primed to analyse a different structural aspect, commonly found in spreadsheets. In addition, they provide a header inference framework which allows to employ a combination of the mentioned algorithms. Using this framework, Abraham and Erwig proceed by evaluating and optimizing different weighted combinations of header detection algorithms. In later work, [2007] Abraham and Erwig utilize their optimized header inference process by combining it with previous work into unit systems for spreadsheets [Erwig and Burnett, 2002]. The resulting system was implemented into a tool dubbed *UCheck*. The tool is provided as an add-in for *Excel* which communicates with a header/unit inference engine implemented in Haskell. Once the system is put into action, it automatically infers headers and

assigns and infers units. The tool indicates cells which contain detected unit errors via a red background color. Evaluation by Abraham and Erwig indicates that the header inference process rarely produces incorrect header assignments. Moreover, even in cases where incorrect header inference occurs, the follow-up unit inference does not report any illegal errors.

Another approach for unit checking of spreadsheets was proposed by Coblenz *et al.* in [2005]. With *SLATE*, they introduced a spreadsheet system of their own. *SLATE* accurately handles unit dimensions and in addition adopts the more abstract notion of units popularized by Erwig and Burnett [2002] in form of labels. Using this system, value cells can be annotated both with units like \$ and kg as well as labels like "apples" and "oranges". A cell value depicting the per kg price of oranges may look like this: $\$0.96/kg$ $(oranges)$. Dimensions as well as labels need to be entered manually for each cell. To facilitate unit- and label-checking, each spreadsheet requires a unit context which defines usable base units and a label context which defines the hierarchical relationship between different supported labels. Dimension inference within the system follows the mathematical model. Label and unit inference follow a similar approach to the one presented in [Erwig and Burnett, 2002]. The result of these inference processes is supplied to the labels of formula cells. Explicit error-checking capabilities are missing in the spreadsheet system. Nevertheless, users may detect errors themselves by analysing the unit- and dimension-information contained in the updated formula labels. The effectiveness of *SLATE* in comparison to other spreadsheet systems remains unclear, as no relevant evaluation has been conducted.

In 2008, Chambers and Erwig proposed an approach to automatically detect dimension errors in spreadsheets [Chambers and Erwig, 2008]. Their definition of dimensions is similar to the one presented in [2004]. Chambers and Erwig's dimension analysis approach revolves around so-called "base dimensions". Each base dimension is related to a base unit of measurement (e.g. length or mass). Base dimensions may also occur with an applied exponent and be combined with other base dimensions. This allows for the definition of complex and composite dimensions. For example, velocity can be defined as $\{length, time^{-1}\}$. A set of 9 valid base units is available. Each of these base units features a specific default unit (e.g. meter(m) for the base unit length). Conversion factors allow for automatic

coercions to other units of measurement for the same base unit. The unit-checking process follows a specific sequence: First, header inference based on a previously presented approach [Abraham and Erwig, 2007] generates header information for further processing. In a second step, dimensions are derived by analysis of the labels contained in identified header cells. Lastly, the resulting dimension information is propagated and dimension inference applied to value- and formula cells. In cases where dimension information is missing due to failed label analysis, the context provided by formulas can be used as basis to infer the missing dimensions. The system is able to discern two types of errors: *Colliding dimensions* indicate that dimensions inferred for formulas do not match the expected dimension for this cell. *Invalid dimensions* indicate that inferred dimensions violate pre-set constrains (e.g. inference of the dimension kg^3). Cells that contain detected errors are highlighted via application of a predefined background colour and a descriptive note. Chambers and Erwig [2009] refined their approach and implemented a the resulting system into an extension for *Microsoft Excel*. The paper also features an in-depth evaluation of the tool based on the EUSES spreadsheet corpus [Fisher and Rothermel, 2005]. According to this evaluation, label analysis and dimension inference worked reliably. Header inference was still the most problematic step, as labelling practices within spreadsheets vary. Nevertheless, dimension errors were found in almost half of the spreadsheets selected for evaluation.

Lastly, in 2010, Chambers and Erwig proceeded in combining purely label-based and dimension-based reasoning into one system [Chambers and Erwig, 2010]. For a given worksheet, the system attempts to identify a vertical and horizontal label axis. Label interpretation then allows to discern which of those axes can be utilized for dimension interference. In cases where an axis features no dimension information, the system falls back to the label-based analysis approach described in [Abraham and Erwig, 2007]. For cases where both label- and dimension-information is available, Chambers and Erwig propose a variety of different approaches to combine both analysis processes. However, they reason that the best approach is to determine both the labels and dimensions of an axis wherever possible. This approach guarantees to utilize all structural information that label checking can provide without loosing the merit of dimension checking. Indeed, evaluation of various analysis combinations by the authors indicates that the fully integrated approach of combining both systems

yields the highest number of detected errors. However, the integrated system also features a higher number of false positives than each singular analysis approach. In conclusion, Chambers and Erwig reason that, despite the drawbacks, a system integrating label- and dimension-analysis is able to detect more errors and thus proves to be more useful to users.

3.2 Common Concepts

Many of the above mentioned approaches are quite similar to each other or can even be regarded as derived work. Consequently, many of those approaches share a number of common base concepts and methodologies. These common concepts include:

- Identification of cell headers.
- Assignation, combination, and propagation of units.
- Inference, combination, and propagation of dimensions.
- User input and report of detected unit- and dimension errors.

In the following subsections, we present each of these concepts in greater detail. In addition, we explain the basic idea and result of the respective steps in the common work-flow.

To explain how these concepts work in practice, we provide an example in Figure 3.2. This example depicts a spreadsheet which calculates the price of a specific amount of different brands of coffee, based on the price and weight of their respective custom packaging. Both, value and formula views of the example spreadsheet, are available in Subfigures 3.2a and 3.2b. The labels and dimensions resulting of the respective inference processes for formulas are depicted in Subfigures 3.2c and 3.2d.

3.2.1 Header Inference

Header inference denotes the process of identifying a set of headers for each cell within a spreadsheet. A header within a unit-checking system is a cell which provides a unit to its dependent cells. Indeed, a cell can be a header for an arbitrary number of

	A	B	C	D	E	F
1	Coffee	Price	Weight	Price per weight	Standard weight	Total Price
2	Jacobs	260	25	10.4	100	1040
3	HAG	360	30	12	100	1200
4	Illy	240	25	265	100	26500
5						1040

(a) Value view.

	A	B	C	D	E	F
1	Coffee	Price	Weight	Price per weight	Standard weight	Total Price
2	Jacobs	260	25	=B2/C2	100	=D2*E3
3	HAG	360	30	=B3/C3	100	=D3*E3
4	Illy	240	25	=B4+C4	100	=D4*E4
5						=min(F2:F4)

(b) Formula view.

	A	B	C	D	E	F
1	Coffee	Price	Weight	Price per weight	Standard weight	Total Price
2	Jacobs	Price & Jacobs	Weight & Jacobs	Price & Jacobs	Standard weight & Jacobs	ERROR
3	HAG	Price & HAG	Weight & HAG	Price & HAG	Standard weight & HAG	HAG
4	Illy	Price & Illy	Weight & Illy	(Price | Weight) & Illy	Standard weight & Illy	Illy
5						ERROR

(c) Labels, after unit inference.

	A	B	C	D	E	F
1	Coffee	Price	Weight	Price per weight	Standard weight	Total Price
2	Jacobs	$	kg	$ / kg	kg	$
3	HAG	$	kg	$ / kg	kg	$
4	Illy	$	kg	ERROR: $ + kg	kg	$
5						$

(d) Dimensions, after dimension inference.

Figure 3.2: An example worksheet, demonstrating common concepts used within unit-checking approaches. The table depicts a comparison of different coffee brands. A blue border indicates cells inferred as header cells. A yellow background indicates cells containing errors.

cells and can itself have an arbitrary number of headers. Chambers and Erwig [2010] provide a formal definition for the binary relation produced by header inference following the form: $H \subseteq A \times A$. Based on this definition, $(a, a') \in H$ indicates that a' is a header of the cell a.

Within the example depicted in Figure 3.2, header cells are indicated via a blue border. Cells B1 to F1 are identified as headers for their respective columns. Cells A2 to A4 are identified as headers for their respective rows. A1 is identified as header for cells A2 to A4, forming a dependent header relationship.

While Erwig and Burnett already realized the necessity of header inference in their initial approach [Erwig and Burnett, 2002], in-depth consideration of concrete header inference mechanics was omitted at the time. Nevertheless, they did supply some likely starting points for such processes: *Predefined unit information* (e.g. the fact that the label June denotes a month), *Formatting* of specific parts of a spreadsheet, and *Spatial & content analysis* of the supplied spreadsheet. Moreover, in a follow-up work [Abraham and Erwig, 2004], they addressed this problem by presenting a framework for implementation and evaluation of header inference algorithms as well as a collection of spatial-analysis algorithms for header inference. Those algorithms numbered: *Fence Identification*, *Content-Based Cell Classification*, *Region-Based Cell Classification*, and *Footer-to-Core Expansion*. Each of these algorithms identifies a specific set of cells within the spreadsheet with a specific certainty to be headers for specific other cells. Evaluation of the test system indicates that a weighted combination of each of the presented algorithms yields the optimal results for the test data. The proposed techniques were further optimized and resulted in the implementation of the *UCheck* tool, described in [Abraham and Erwig, 2007]. Since then, the proposed process for header inference can be regarded as the de-facto standard and was used repeatedly [Chambers and Erwig, 2008] [Chambers and Erwig, 2009] [Chambers and Erwig, 2010].

Other approaches regarding unit-checking of spreadsheets proposed in [Ahmad *et al.*, 2003], [Antoniu *et al.*, 2004], and [Coblenz *et al.*, 2005] each omitted in-depth investigation of header inference processes. Instead, those approaches provided user interfaces which allowed for cells to be manually annotated with units.

3.2.2 Label-based Unit Inference

The concept of label-based unit inference was introduced by Erwig and Burnett in [2002]. According to their concept, each value within a spreadsheet defines a unit. However, only a subset of these units provides meaningful header information for other cells. This relationship is established either by use of automatic header inference, or by manual annotation by users. Header cells define so-called "simple units" (e.g. *Coffee*, *Espresso*, and *July*). Simple units may be headers of other simple units (e.g. *Coffee[Espresso]*). Such relations construct a hierarchy of dependent units. Simple units may be combined into complex units in two distinct ways: Firstly, a cell can depend on more than one header. The unit of such a cell is a combination of the units of each related header cell. Secondly, formula operators combine units of referenced cells. Examples for complex units are *Coffee[Espresso] & July* and *Coffee[Espresso] | Coffee[Decoffeinated]*. During unit inference, formula operators combine units of referenced operand cells. If the operation is inapplicable due to mismatching operand-units, this circumstance is detected as unit error. Likewise, if the unit resulting from the combination of multiple referenced units is not sound, this circumstance is detected as unit error. While most unit inference system follow this basic concept, the exact unit-inference rules and soundness-criteria for resulting units vary.

An example of a unit-error is contained within the spreadsheet in Figure 3.2. Cell F2 contains the formula D2*E3. Cell D2 has the unit *Price per weight & Jacobs* whereas E3 has the unit *Standard weight & HAG*. Both cells exclusively contain unit factors which are different from the factors of the other cell. Consequently, unification of the units is impossible. This would be detected as an error within unit-checking processes. In comparison, Cell F3 contains the formula D3*E3. D2 has the unit *Price per weight & HAG*. E2 has the unit *Standard weight & HAG*. Both referenced cells feature the unit factor *HAG*. Consequently, these units can be unified successfully, retaining the *HAG* portion. No error would be reported for this formula.

3.2.3 Dimension Inference

Dimension inference is a more elaborate technique of label-based unit inference. This technique relies on the more classical interpretation of the term unit in regard to a value: a unit of measurement in a specific dimension. Antoniu *et al.* [Antoniu *et al.*, 2004] and Chambers and Erwig [2008] both presented similar approaches for dimension-checking of spreadsheets. Both approaches depict the use of a base unit expanded by an exponent. Antoniu *et al.* [2004] allow arbitrary names for base units while Chambers and Erwig [2008] only allow a set of 9 predefined base units to be used. Both allow the construction of more complex units combining base units (e.g. $m^1 s^{-1}$). Depending on the approach, dimension information of value- and formula cells is either assigned by users manually, or header inference is used to derive dimensions automatically. During dimension inference, formulas are processed bottom-up, combining units of referenced cells to form more complex units. The resulting units are then checked against the expected dimensions for each formulas' values. If the inferred unit does not match the expectation provided by annotation or header analysis, an error is reported. In addition, Chambers and Erwig suggest to sanity-check inferred units: Each inferred unit is checked against a list of known, meaningful units in order to detect outliers which are mathematically correct but do not occur in a scientific context (e.g. kg^3).

An example of a dimension-error is contained within the spreadsheet depicted in Figure 3.2. Cell D4 contains the formula B4+C4. Dimension inference identifies $\$^1$ and kg^1 for the dimensions of the referenced cells. However, addition of different base units is prohibited. Consequently, Cell D4 is indicated to contain a dimension error. Note that conventional label-based unit-checking would not be able to identify this error. Both referenced cells contain the label "Illy". Thus, unification of those cells based on conventional unit inference results in a valid unit.

3.2.4 User Interaction

Unit-checking systems operate by processing provided unit information and informing users about detected errors. Thus, such systems need to define interfaces which allow a user to provide unit information to the system, and interfaces which allow

the system to provide its results to the user. In the following subsection, we provide an overview of common user-interface concepts employed in various unit-checking approaches.

In order to process and validate a spreadsheet, unit-checking systems need to know the units of value cells and expected units of formula cells. Over time, different methods have been established to provide this information. In their initial proposal, Erwig and Burnett simply assumed for this information to be available [2002]. The first systems to actually implement unit-checking required users to annotate their spreadsheets manually. The tools presented in [Ahmad *et al.*, 2003] and [Antoniu *et al.*, 2004] both provide a simple user interface which allows to manually enter cell range-unit pairs. The spreadsheet environment SLATE [Coblenz *et al.*, 2005] allows for annotation of values directly within their respective cells. More recent approaches by Chambers and Erwig [2008] [2009] [2010] utilize header inference methods introduced in [Abraham and Erwig, 2004] and [Abraham and Erwig, 2007]. Consequently, no explicit user input is required beside the definition of row and column headers.

Users need to be adequately informed about the unit-errors detected by unit-checking processes in order to react appropriately. A combination of highlighting background-colors and explaining notes established itself as de-facto standard within the community for that purpose. Background-colors indicate the presence of a detected error within a cell. Usually, different color-hues are used to represent the severity or type of detected errors. In addition, most approaches add a note to each indicated cell, explaining the detected error in greater detail. Ahmed *et al* [2003], Antoniu *et al.* [2004], Abraham and Erwig [2007], and Chambers and Erwig [2008] [2009] [2010] all employ some variation of this combination, whereas Abraham *et al.* [2004] employ background-colors only. The use of arrows is another, commonly employed concept. Antoniu *et al.* [2004] display arrows to indicate the connection between a faulty formula cell and cells which are referenced within the formula. Abraham and Erwig employ arrows to visualize header dependencies within their systems [2004] [2007]. Lastly, the spreadsheet system proposed by Coblenz *et al.* simply annotates formula cells with the result of its unit inference process. Users may validate their formulas manually by checking these annotations.

3.3 Derived Concepts

Unit- and dimension-checking enables users to detect and correct semantic errors within formulas. The basic concepts employed by these processes can be used within more specific or generally different approaches to improve spreadsheet quality. Indeed, various approaches were proposed which further expand on those notions. In the following section, we exemplary present a few of them.

Type systems expand the notion of units and dimensions employed by the previously presented static checking approaches. In particular, they define type expectations for the parameters and the result types of formula cells. Abraham and Erwig proposed their approach for a spreadsheet type system in [Abraham and Erwig, 2006b]. Following their approach, they specify the types of a formula as so-called "function types". Function types limit allowed argument types of formulas by defining the types that are expected for a formula's referenced cells. The "result type" of a formula is defined by the result type of the outermost operation of the formula. Based on these prerequisites, a "cell type" can be defined as a pair containing the formula's result type and possible type conflicts based on the formula's references. Lastly, cell types can be used to define "spreadsheet types" which can be utilized to calculate the type correctness for different parts of a spreadsheet. Based on these definitions, Abraham and Erwig proposed a type inference algorithm to detect type errors within a spreadsheet. In addition, the same type inference process can be expanded to export templates and models of an existing spreadsheet.

Content- and structure-information of a spreadsheet can be used to infer its so-called *Spreadsheet Templates*. Template inference processes the structure and content information of an existing worksheet in order to determine which content type is applicable for each row and column of the worksheet. Generated worksheet templates can be used to guarantee that columns and rows of the worksheet may only be expanded by new cells which pass the determined criteria. Thus, range, reference, and type errors can be prevented automatically. Abraham and Erwig describe a spreadsheet template system as well as a matching template inference algorithm in [Abraham and Erwig, 2006a].

The combination of spreadsheet templates with the notion of types allows to specify so-called "Spreadsheet Models". A model is an abstract definition of each permitted column and row, defining worksheet-templates of a spreadsheet. Thus, a model can be determined by extracting the template of each worksheet of the spreadsheet in question. Likewise, a spreadsheet model can be used to determine a set of templates. Thus, a concrete spreadsheet containing data can be considered as an instance of the abstract model its based on. By enforcing the definitions of a spreadsheet model, only cell-updates may be allowed which never produce any reference- or type errors. Abraham and Erwig demonstrate how to create and utilize spreadsheet models based on their previous work in [Abraham and Erwig, 2006b]

4. Conclusions

The main purpose of this study was to provide an in-depth review of the state of the art of specific static analysis techniques for spreadsheet environments. In specific, we presented and elaborated on an extensive catalogue of spreadsheet smells introduced by the scientific community. In addition, we outlined the history and basic concepts of label-based unit- and dimension-checking approaches for spreadsheets.

We conclude that both are adequate techniques to improve overall spreadsheet quality. Spreadsheet smells provide meaningful feedback about specific quality properties. In most approaches, cells which contribute to detected smells are indicated visually and decorated with a note which further explains the issue. Either manual or automatic refactoring can then be applied to remove those issues, raising spreadsheet quality as a consequence. Coincidentally, provided labels may convey a general understanding of quality aspects to users who manually refactor the indicated issues. This expanded comprehension of quality properties may aid spreadsheet users further along during the maintenance or creation of further spreadsheets. Unit- and dimension-checking of spreadsheets provides a more tangible improvement to spreadsheet quality. Either manual annotation or header inference provides unit-information for value and formula cells. Based on that information, approaches based on these techniques indicate cells whose formulas compute erroneous or unexpected units. Consequently, formulas are detected which may be syntactically correct, but infringe on the semantic intention of the spreadsheet. In addition, users may adapt a more considerate approach to spreadsheet structure, as they are required to either provide meaningful information to allow for automated header inference or even manually annotate value cells with expected units.

Both approaches also suffer from some drawbacks. Spreadsheet smells indicate potential quality impairments rather than specific errors. However, numeric thresh-

olds for metric values at which cells are indicated as smelly are statically defined and ignore the current context of the spreadsheet in question. Moreover, spreadsheet smells and the quality aspects they represent do not exist in a vacuum. Actions to improve one quality aspect usually have a negative effect on another quality aspect as well. Lastly, removal of spreadsheet smells frequently requires a considerable amount of re-structuring within the spreadsheet. Manual refactoring therefore is often unattractive for end-users, or even leads to the introduction of new errors to the spreadsheet. In regard to unit- and dimension-checking, the main drawback is the initial information requirement. Users either need to manually annotate cells with unit or dimension-information, or at least define row- and column-headers viable for automated header inference. Furthermore, more elaborate techniques which utilize both unit- and dimension-information to deduce errors also feature higher false-positive detection rates.

As mentioned in Subsection 2.2, inter-worksheet smells are missing a counterbalance for optimization. This is a novel finding and should be further explored. In specific, one or more concrete metrics indicating at which point spreadsheet logic should be transferred to a new worksheet are of interest.

Future work into the field of static spreadsheet analysis should focus on expanding the understanding about characteristics of and interactions between spreadsheet smells. New smells should be introduced, either based on further existing code smells or using spreadsheet-specific quality measurements. Further inter-worksheet smells to counteract the existing set would be of particular interest. Moreover, improved tool support to refactor the quality issues identified by smells would be favourable. As for unit- and dimension-checking, future work should abandon the requirement for manual annotation in general. Instead, the existing methods for header inference should be refined and expanded on, as even recent approaches identify this step as weak spot. In addition, processes to automatically recognize or prohibit the detection of false positives could greatly improve the results of existing approaches.

List of Figures

List of Tables

References

[Abraham and Erwig, 2004] Robin Abraham and Martin Erwig. Header and unit inference for spreadsheets through spatial analyses. In *IEEE Symposium on Visual Languages and Human Centric Computing, 2004. VL/HCC 2004*, pages 165–172, September 2004.

[Abraham and Erwig, 2006a] Robin Abraham and Martin Erwig. Inferring templates from spreadsheets. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 182–191, New York, NY, USA, 2006. ACM.

[Abraham and Erwig, 2006b] Robin Abraham and Martin Erwig. Type inference for spreadsheets. In *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '06, pages 73–84, New York, NY, USA, 2006. ACM.

[Abraham and Erwig, 2007] Robin Abraham and Martin Erwig. UCheck: A spreadsheet type checker for end users. *Journal of Visual Languages & Computing*, 18(1):71–95, February 2007.

[Abreu *et al.*, 2014a] Rui Abreu, Jácome Cunha, Joao Paulo Fernandes, Pedro Martins, Alexandre Perez, and Joao Saraiva. Smelling faults in spreadsheets. In *Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution, ser. ICSME*, volume 14, 2014.

[Abreu *et al.*, 2014b] Rui Abreu, Jácome Cunha, Joao Paulo Fernandes, Pedro Martins, Alexandre Perez, and Joao Saraiva. FaultySheet detective: When smells meet fault localization. In *2014 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 625–628, September 2014.

[Ahmad *et al.*, 2003] Yanif Ahmad, Tudor Antoniu, Sharon Goldwater, and Shriram Krishnamurthi. A type system for statically detecting spreadsheet errors. In *18th IEEE International Conference on Automated Software Engineering, ASE 2003. Proceedings*, pages 174–183, October 2003.

[Antoniu *et al.*, 2004] Tudor Antoniu, Paul A. Steckler, Shriram Krishnamurthi, Erich Neuwirth, and Matthias Felleisen. Validating the unit correctness of spreadsheet programs. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 439–448, Washington, DC, USA, 2004. IEEE Computer Society.

[Badame and Dig, 2012] Sandro Badame and Danny Dig. Refactoring meets spreadsheet formulas. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 399–409, September 2012.

[Burnett and Erwig, 2002] Margaret Burnett and Martin Erwig. Visually customizing inference rules about apples and oranges. In *Proceedings of the IEEE 2002 Symposia on Human Centric Computing Languages and Environments (HCC'02)*, HCC '02, pages 140–, Washington, DC, USA, 2002. IEEE Computer Society.

[Chambers and Erwig, 2008] Chris Chambers and Martin Erwig. Dimension inference in spreadsheets. In *IEEE Symposium on Visual Languages and Human-Centric Computing, 2008. VL/HCC 2008*, pages 123–130, September 2008.

[Chambers and Erwig, 2009] Chris Chambers and Martin Erwig. Automatic detection of dimension errors in spreadsheets. *Journal of Visual Languages & Computing*, 20(4):269–283, August 2009.

[Chambers and Erwig, 2010] Chris Chambers and Martin Erwig. Reasoning about spreadsheets with labels and dimensions. *Journal of Visual Languages & Computing*, 21(5):249–262, December 2010.

[Chiang and Miller, 2008] Fei Chiang and Renée J. Miller. Discovering data quality rules. *Proceedings of the VLDB Endowmen, Proc. VLDB Endow.*, 1(1):1166–1177, August 2008.

[Coblenz *et al.*, 2005] Michael J. Coblenz, Andrew J. Ko, and Brad A. Myers. Using objects of measurement to detect spreadsheet errors. In *2005 IEEE Symposium*

on Visual Languages and Human-Centric Computing, pages 314–316, September 2005.

[Cunha *et al.*, 2012a] Jácome Cunha, Joao Paulo Fernandes, Pedro Martins, Jorge Mendes, and Joao Saraiva. SmellSheet detective: A tool for detecting bad smells in spreadsheets. In *2012 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 243–244, September 2012.

[Cunha *et al.*, 2012b] Jácome Cunha, João P. Fernandes, Hugo Ribeiro, and João Saraiva. Towards a catalog of spreadsheet smells. In *Computational Science and Its Applications – ICCSA 2012*, number 7336 in Lecture Notes in Computer Science, pages 202–216. Springer Berlin Heidelberg, January 2012.

[Erwig and Burnett, 2002] Martin Erwig and Margaret Burnett. Adding apples and oranges. In *Practical Aspects of Declarative Languages*, number 2257 in Lecture Notes in Computer Science, pages 173–191. Springer Berlin Heidelberg, 2002.

[Fisher and Rothermel, 2005] Marc Fisher and Gregg Rothermel. The EUSES spreadsheet corpus: A shared resource for supporting experimentation with spreadsheet dependability mechanisms. In *Proceedings of the First Workshop on End-user Software Engineering*, WEUSE I, pages 1–5, New York, NY, USA, 2005. ACM.

[Fowler, 1999] Martin Fowler. *Refactoring: improving the design of existing code.* Pearson Education India, 1999.

[Hermans *et al.*, 2011] Felienne Hermans, Martin Pinzger, and Arie van Deursen. Supporting professional spreadsheet users by generating leveled dataflow diagrams. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 451–460, New York, NY, USA, 2011. ACM.

[Hermans *et al.*, 2012a] Felienne Hermans, Martin Pinzger, and Arie van Deursen. Detecting and visualizing inter-worksheet smells in spreadsheets. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 441–451, Piscataway, NJ, USA, 2012. IEEE Press.

[Hermans *et al.*, 2012b] Felienne Hermans, Martin Pinzger, and Arie van Deursen. Detecting code smells in spreadsheet formulas. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 409–418, September 2012.

[Hermans *et al.*, 2013] Felienne Hermans, Ben Sedee, Martin Pinzger, and Arie van Deursen. Data clone detection and visualization in spreadsheets. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 292–301, Piscataway, NJ, USA, 2013. IEEE Press.

[Hermans *et al.*, 2014] Felienne Hermans, Martin Pinzger, and Arie van Deursen. Detecting and refactoring code smells in spreadsheet formulas. *Empirical Software Engineering*, pages 1–27, February 2014.

[Jannach *et al.*, 2014] Dietmar Jannach, Thomas Schmitz, Birgit Hofer, and Franz Wotawa. Avoiding, finding and fixing spreadsheet errors – a survey of automated approaches for spreadsheet QA. *Journal of Systems and Software*, 94:129–150, August 2014.

[Levenshtein, 1966] VI Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. In *Soviet Physics Doklady*, volume 10, page 707, 1966.

[Myers *et al.*, 1992] Brad A. Myers, David Canfield Smith, and Bruce Horn. Languages for developing user interfaces. pages 343–366. A. K. Peters, Ltd., Natick, MA, USA, 1992.

[Panko and Port, 2012] Raymond R. Panko and Daniel N. Port. End user computing: The dark matter (and dark energy) of corporate IT. In *2012 45th Hawaii International Conference on System Science (HICSS)*, pages 4603–4612, January 2012.