Refactoring Smelly Spreadsheet Models

Pedro Martins and Rui Pereira

HASLab / INESC TEC & Universidade do Minho, Portugal
{prmartins,ruipereira}@di.uminho.pt

Abstract. Identifying bad design patterns in software is a successful and inspiring research trend. While these patterns do not necessarily correspond to software errors, the fact is that they raise potential problematic issues, often referred to as code smells, and that can for example compromise maintainability or evolution.
The identification of code smells in spreadsheets, which can be viewed as software development environments for non-professional programmers, has already been the subject of confluent researches by different groups. While these research groups have focused on detecting smells on concrete spreadsheets, or spreadsheet instances, in this paper we propose a comprehensive set of smells for abstract representations of spreadsheets, or spreadsheet models. We also propose a set of refactorings suggesting how spreadsheet models can become simpler to understand, manipulate and evolve. Finally we present the integration of both smells and refactorings under the MDSheet framework.

Keywords: Spreadsheets, Smells, Model-Driven Engineering, ClassSheets

1 Introduction

For many people, a spreadsheet system is the programming language and programming environment of choice. A spreadsheet programmer is often referred to as an *end user* and is a secretary, an accountant, a teacher, a student, an engineer, or any other person that is not a professional programmer and simply wants to solve a problem [31]. Although spreadsheets were introduced to help end users to perform simple mathematical operations, they quickly evolved into powerful software systems heavily used in industry by professional programmers to process complex and large data. For example, spreadsheets are often used in industry as a simple mechanism to adapt data produced by one system to the format required by a different one.

Regardless of their huge commercial success and wide acceptance, spreadsheets are also known for being highly error-prone! This is supported by both research studies [33,36,37,32] and errors reported in the media[1].

These facts suggest that spreadsheets are a particularly significant target for the application of software engineering principles. Surprisingly, only recently the

[1] A list of spreadsheets horror stories is maintained by the *European Spreadsheet Risks Interest Group* at http://www.eusprig.org/horror-stories.htm.

research community started investigating the use of software engineering techniques in spreadsheets [24]. In software engineering, models were widely adopted as a suitable abstraction mechanism to specify/express complex software. Naturally, spreadsheets followed this trend, namely in the use of model-driven software development techniques [23,19,5,3,21], software refactoring and evolution techniques [20,10,22,14,6], and software metrics and quality models [7,16,8,15].

In this paper we present techniques to detect and eliminate *spreadsheet smells* in the context of model-driven spreadsheet development. A model-driven spreadsheet smell is not an error, but an indication of a poorly designed spreadsheet model. As a consequence spreadsheet smells can be used to improve a spreadsheet model's maintainability and quality. After presenting ClassSheets and a motivational example in Section 2 we present our contributions:

- We adapt a catalog of spreadsheet smells to work on model-driven spreadsheet. That is to say that for each smell defined at the spreadsheet data level we define an equivalent at the ClassSheet level. (Section 3);
- For each smell on ClassSheet models that we propose, we present a possible refactoring or set of refactorings to eliminate it (Section 4). The refactorings are expressed as model evolutions. Thus eliminating smells at the model level automatically eliminates smells at the data level;
- We automated smell detection with an implementation under the MDSheet framework (Section 5.1);
- We also implemented automated refactorings under MDSheet (Section 5.2);

We conclude this paper exposing related work in Section 6 and drawing some conclusions in Section 7.

2 ClassSheets– Models for Spreadsheets

Before presenting spreadsheet model smell detection, let us introduce as a running example model-driven spraedsheet. Figure 1 presents a spreadsheet used to grade students' scores. This spreadsheet contains information on various *Marks*,

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
1	Marks	Exam	First			Exam	Second			Exam	Third			⋯		
2		Part 1	Part 2	Part 3	Mark	Part 1	Part 2	Part 3	Mark	Part 1	Part 2	Part 3	Mark	⋯	Final Mark	Grade
3		25	25	50	10	10	10	80	5	10	10	80	5	⋯	100	{A,...,F}
4	Student													⋯		
5	A	50	70	20	4	35	10	80	3.425	40	35	70	3.175	⋯	51.75	C
6	B	65	20	35	3.875	20	45	35	1.725	60	20	40	2	⋯	39.375	D
7	C	90	100	85	9	75	95	80	4.05	80	85	90	4.425	⋯	89.25	A
8	D	100	45	75	7.375	40	60	50	2.5	50	75	55	2.825	⋯	65.125	B
9	E	35	60	80	6.375	60	50	55	2.75	40	20	70	3.1	⋯	62.875	B
10	F	70	35	35	4.375	45	45	60	2.85	30	60	50	2.45	⋯	46.375	C
11	G	95	80	80	8.375	70	80	80	3.95	70	90	85	4.2	⋯	83.875	A
12	H	75	55	65	6.5	20	50	20	1.15	50	50	65	3.1	⋯	63.5	B
13	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋯	⋮	⋮
14														⋯		

Fig. 1. Instance of a grading spreadsheet.

Exams, and *Students*. Each *Exam* has 3 parts, each having its own weight value.

Every *Student* has a mark from each exam, calculated by a formula using the weights of each part, and the value the student obtained in the respective part. A student's final mark is the average mark across all exams.

This spreadsheet business logic can be abstracted and represented by a model. In this paper we consider ClassSheet model [23,12]: a high-level object-oriented abstract representation for spreadsheets. They can be compared to UML class diagrams [18], but for spreadsheets and with layout specification. Indeed they are formed by classes and attributes, which we will explain in the following paragraphs using the marks spreadsheet example. Figure 2 is the ClassSheet model to abstract and specify this grading spreadsheet. There are 3 classes,

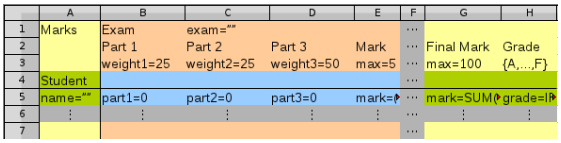

Fig. 2. ClassSheet model of a grading spreadsheet.

Marks, **Student**, and **Exam**. Each **Student**, represented by row `5`, has an attribute termed *name*, which is set in cell `A6`. Each *attribute* in a ClassSheet is defined by a string (e.g., name), the equal sign, and by a default value or a formula. Note that students expand vertically, that is, each student is added after the previous one in the next row. This is encoded in the model by the ellipsis shown in row `6`. Each **Exam** has an attribute with the weight for each of the 3 parts, *weight1*, *weight2*, and *weight3* respectively, and expands horizontally. This is indicated by the ellipsis in column `F`. The relationship between a **Student** and an **Exam** gives us 4 attributes. The first 3, *part1*, *part2*, and *part3*, state the mark the student obtained in the specific part. The last attribute, *mark*, states the mark the student obtained on the exam. This *mark* formula is defined as:

```
mark=(weight1*part1 + weight2*part2 + part3*weight3) /
    ↪ SUM(weight1,weight2,weight3) * max /
    ↪ SUM(weight1,weight2,weight3)
```

The **Marks** class has 2 attributes. The first, *mark* (in cell `G5`, only partially), states the final mark a student obtained throughout the exams. The second, *value* (in cell `H5`, hidden by the *mark* formula), states the mark value given (A,B,C,D, or F). These two formulas are defined respectively as:

```
mark=SUM(Exam.mark*max/Exam.max)) / COUNT(Exam.mark)
grade=IF(mark<=(max/5),"F", IF(mark<=(max/5*2),"D",
    ↪ IF(mark<=(max/5*3),"C", IF(mark<=(max/5*4),"B","A"))))
```

Note that these formulas only differ from regular spreadsheet formulas by the use of attributes instead of cell references. This can be compared to the use of *cell names* in standard spreadsheet systems. Also note that some references can be directed to particular attributes as follows `Exam.max`.

3 Spreadsheet Model Smells

The concept of code smell [25] (or simply bad smell) was introduced as a concrete evidence that a piece of software may have a problem. Usually a smell is not an error in the program, but a characteristic that may cause problems understanding the software (for example, a long class in an object-oriented program) and updating and evolving the software. Although they were initially designed for objected-oriented programming, code smells have been adapted for other contexts, including spreadsheets [8,7,28,27]. In this section we survey the code smells defined for spreadsheet formulas introduced in [28]. Moreover, we adapt each of these five smells to the spreadsheet models described in [10,11,13,18,12].

3.1 Multiple Operations

The first smell we present is the *multiple operations* smell. This smell can be considered as present in a spreadsheet if a formula is created using too many operations. Clearly it is important to understand what "too many" exactly means in this scenario, and we will explain it in detail in Section 5.1. Note that this smell can be detected in individual cells.

Let us look at a formula extracted from our running example (from Figure 2), in this case the formula to evaluate the mark of a student for a specific exam:

```
mark=(part1*weight1 + part2*weight2 + part3*weight3) /
    ↪ SUM(weight1,weight2,weight3) * max /
    ↪ SUM(weight1,weight2,weight3)
```

As shown in this formula, ClassSheet formulas are quite similar to the ones written in regular spreadsheet systems. Indeed the operations (operators and formula names) are used in a similar way. Thus, we consider the model smell as it is considered in regular spreadsheets.

In this case we have 9 operations, and as we will see these are indeed too many to have in a formula and still be considered maintainable. Indeed having so many operations in the same cell has several disadvantages: for once, it becomes hard to understand what such a formula does; it also becomes difficult to update or change such a formula.

3.2 Multiple References

Let us consider the same formula shown in the previous smell. This formula has a second quality problem: it uses too many references. When defining ClassSheet models, instead of using regular spreadsheet references such as `A1` or `B2`, the user writes the name of the attributes used formulas. In this case, `part1` or `weight1` are references to the attributes with the same name. This can be compared to the use of *cell names* in standard spreadsheet systems. Thus, in this case the smell adapts very well from plain spreadsheets to the ClassSheet models. In the `mark` formula shown before there are 12 references to other attributes which makes this formula hard to understand and maintain. Finally note that, as with the previous smell, this can be detected for each cell defined by a formula.

3.3 Condition Complexity

The third smell we consider is well known and applicable whenever conditional construction is possible in programming [29]. Thus this is also true for spreadsheets, and for spreadsheet models.

Let us consider the formula to present the *final mark* of the student:

```
value=IF(mark<=20,"F",IF(mark<=40,"D", IF(mark<=60,"C",
    ↪ IF(mark<=80,"B","A"))))
```

This formula is defined using 4 `if` conditions, which again makes it hard to manipulate. Indeed this is true in many other programming languages, but is especially true in spreadsheets as it must be defined in one line without any possible indentation as possible in most regular programming languages.

Since conditional programming is possible in these models, this smell also applies to their formulas. Once again, this smell can potentially occur in each and every cell of the model.

3.4 Long Calculation Chain

This smell appears in spreadsheets that require long paths of cell dereferencing to calculate the values of their formulas. For instance, if a formula has a reference to cell `A1`, and `A1` has a reference to `A2`, and `A2` has a reference to `A3`, then there is a long path that is necessary to run to compute the value of `A1`. This has a considerable impact on the formula's quality since it is necessary to go through several cells to see its arguments and understand it. Also, its result may change if one of its references changes its value, which may be difficult to notice it the path is too long or even comprises of different worksheets.

Let us again look at the formula example of the previous smell. Such a formula depends on the attribute `mark`, which depends on the attribute `part1`. Thus this smell also applies nicely to ClassSheet models. Indeed, instead of normal spreadsheet references we have references to attributes. This also makes it difficult to understand and evolve such formulas.

Finally note that in this case it is not possible to discover this smell only by looking at a cell. It is necessary to follow its references to other attributes recursively.

3.5 Duplicated Formulas

The last smell we adapt for ClassSheet models is *duplicated formulas*. This has similar problems to duplicated code in a regular programming language. In the case of spreadsheets, this occurs in a formula if part of its code is repeated.

Once again considering the `mark` formula, one can see that part of its definition, `SUM(weight1,weight2,weight3)`, appears twice. Thus, this smell also applies to spreadsheet models.

Note that this smell can also occur when considering a range of cells. If part of a formula is repeated through a set of cells, then this repeated part of the formula is also considered duplication. Thus, this smell can occur in individual cells, but also in groups of cells.

3.6 MDSheet with Smell Detection

The smell detection presented in this paper was implemented within MDSheet. In MDSheet's toolbar, a new button is available to evaluate the smells on a model. After clicking the button, the model is annotated with comments that describe the smells present in the annotated cells.

In Figure 3, three cells with smelly attributes were annotated with comments, namely `mark` (in cell `E5`), `mark` (in cell `G5`), and `grade` (in cell `H5`).

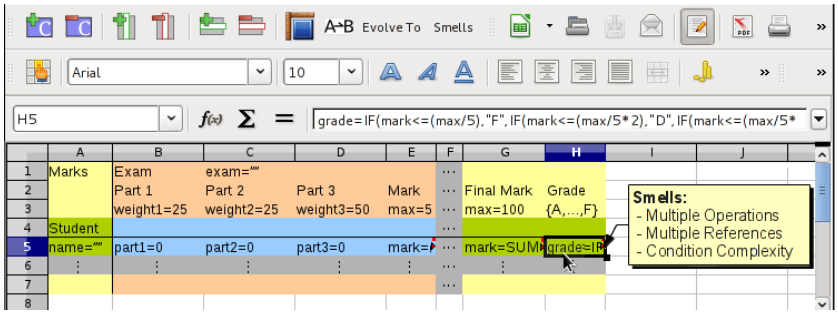

Fig. 3. Smell detection in MDSheet.

4 Refactorings for Spreadsheet Models

In the previous section, we presented a catalog of smells for ClassSheets. The detection of such smells has been implemented in a model-driven spreadsheet environment. Before we present such environment, let us present techniques to automatically eliminate the smells. To this end we rely on *program refactorings* that change a particular piece of code, removing the smell, and not changing the semantics of the code. In this section we present a set of refactorings that can be used to eliminate the smells identified in the previous section. The following refactorings were defined by Hermans et al. [28] in order to remove the smells that they adapted to spreadsheets. We will explain each one and make the necessary adaptation to make them useful for ClassSheets.

4.1 Extract Subformula

This refactoring is intended to extract subformulas from exiting ones. Looking at the arguments at the root of a formula, we can extract new formulas from each argument at the root. This will create new formulas which will be referenced by the original formula.

Although this refactoring has been originally proposed for regular spreadsheet formulas, it can also be applied to models. In the case of models however this process requires the creation of a new attribute for each subformula extracted. These attributes will be created in the same class the subsformulas was extracted from.

4.2 Extract Common Subformula

This is an additional supporting refactoring which can be applied if a formula contains the same subformula multiple times. This refactoring follows the same procedure as *Extract Subformula*.

Once again, this refactoring can be applied to ClassSheets. For example, looking at the `mark` formula one can see `SUM(weight1,weight2,weight3)` repeated twice. Thus such subformula can be extracted, and a new attribute can be created using it: `total_weight=SUM(weight1,weight2,weight3)`. It in now possible to simply reference `total_weight` twice in the formula.

4.3 Merge Branches

When we have a complex `IF` condition, we can sometimes combine multiple branches into one if multiple branches result in the same value. This is a refactoring that is quite transversal to many programming languages, and in particular it is applicable to ClassSheets. For example, the following conditional construction `IF(mark>=3,"PASSED",IF(part3>=5,"PASSED","FAILED"))` can be rewritten as `IF(OR(mark>=3,part3>=5),"PASSED","FAILED")`.

4.4 Merge Formulas

One of the introduced smell is the long calculation chain. This refactoring tries to eliminate it. It can be applied when we detect that there are certain calculation steps in a chain that do not occur in other chains. When this is detected, those two calculation steps are merged into one formula without affecting the other computations as the merged calculation is not being used anywhere else.

As we explained, the long calculation chain is adaptable to the context of spreadsheet models. Moreover, this is also true for this refactoring. In the case of models this can be applied if an attribute is defined by a formula and it is only used in one place. In such cases, such attribute can be removed and the formula integrated into the other computation. In our model both `mark` formulas could be integrated in the formulas that use them. Note this would probably create more smells, and thus should not be applied in this particular situation.

4.5 Other Refactorings

In [28] two more refactorings are presented, namely *Group References* and *Move References*.

Group References can be applied when a spreadsheet cell references a series of adjacent cells, for example `SUM(A1;A2;A3;A4;A5)`. In this case one can restructure this into a lower number of ranges such as `SUM(A1:A5)`. This smell however cannot be applied to ClassSheet as no such range feature is present.

Move References is a refactoring which realocates the cells in a spreadsheet. If a formula is referencing multiple cells in different parts of the spreadsheet, for example `SUM(A1:A5;B5;B19;C4;C7)` one could move the values in `B5`, `B19`, `C4`, and `C7` to `A6` to `A9`. This allows to rewrite the formula as `SUM(A1:A9)`. Again, given that no ranges are possible in ClassSheets this refactorings does not apply.

4.6 Relationship Between Smells and Refactorings

As shown in Table 1, several of the refactorings just presented can be applied to remove different ClassSheet smells. For instance, the extraction of subformula can be used to remove all smells except the long calculation chain. The way these refactorings are applied to remove the smells is detailed in Section 5.

Table 1. ClassSheet smells and refactorings that can be applied to reduce them.

	Multiple Operations	Multiple References	Condition Complexity	Long Calculation Chain	Duplicated Formulas
Extract Subformula	✓	✓	✓		✓
Extract Common Sub.	✓		✓		
Merge Branches			✓		
Merge Formulas				✓	

5 Model-Driven Spreadsheet Framework

In the previous sections we have introduced the concept of *smell* and *refactoring* adapted to spreadsheet models, that is, to ClassSheets. In section 5.1 we will present how such smells can be detected under the MDSheet framework [13,11]. Moreover, in section 5.2 we will explain how the refactorings presented before can be incorporated under the same framework.

We choose this framework because it already integrates several model-driven spreadsheet features, including evolution [10], or data querying [34,9,4,17]. Moreover, given its modular design it allows for easy integration of new functionality, as the one we propose in this paper.

5.1 Detection of Smells in Spreadsheet Models

In this section we present in detail the detection of smells in ClassSheet models. As previously introduced, some of the smells depend on software metrics of certain artifacts that appear in formulas, e.g. the number of operations or the number of references. Following the approach presented in [28], each smell is classified as having *low*, *moderate*, or *high* risk, depending on these metrics. Obviously, the best case scenario is when no smell is present meaning there is no risk. This is the first step our implementation must handle. Thus next we present a Haskell [35] data type that represents these four options:

$$\textbf{data}\ \mathit{Risk} = \mathit{None} \mid \mathit{Low} \mid \mathit{Moderate} \mid \mathit{High}$$

In order to find a smell, and the associated risk, we defined a set of functions that operate on spreadsheet models. Their signatures are presented next:

$$\begin{aligned} &\mathit{smellMultipleOperations} &&:: \mathit{Model} \rightarrow [(\mathit{Cell}, \mathit{Risk})] \\ &\mathit{smellMultipleReferences} &&:: \mathit{Model} \rightarrow [(\mathit{Cell}, \mathit{Risk})] \end{aligned}$$

$$
\begin{aligned}
&\mathit{smellConditionComplexity} &&:: \mathit{Model} \to [(\mathit{Cell}, \mathit{Risk})]\\
&\mathit{smellLongCalculationChain} &&:: \mathit{Model} \to [(\mathit{Cell}, \mathit{Risk})]\\
&\mathit{smellDuplicatedFormulas} &&:: \mathit{Model} \to [(\mathit{Cell}, \mathit{Risk})]
\end{aligned}
$$

These functions receive the model under consideration as an argument and return the list of cells of the models that contain attributes, and the respective risk.

The risk of a smell is obtained by evaluating the related metric on the model and then mapping it to the corresponding risk as set by the thresholds.

$$
\begin{aligned}
&\mathit{smellMultipleOperations} &&= \mathit{map}\ (\mathit{id} \times \mathit{risk}\ 4\ 5\ 9) &&\circ \mathit{metricMultipleOperations}\\
&\mathit{smellMultipleReferences} &&= \mathit{map}\ (\mathit{id} \times \mathit{risk}\ 3\ 4\ 6) &&\circ \mathit{metricMultipleReferences}\\
&\mathit{smellConditionComplexity} &&= \mathit{map}\ (\mathit{id} \times \mathit{risk}\ 2\ 3\ 4) &&\circ \mathit{metricConditionComplexity}\\
&\mathit{smellLongCalculationChain} &&= \mathit{map}\ (\mathit{id} \times \mathit{risk}\ 4\ 5\ 7) &&\circ \mathit{metricLongCalculationChain}\\
&\mathit{smellDuplicatedFormulas} &&= \mathit{map}\ (\mathit{id} \times \mathit{risk}\ 6\ 9\ 13) &&\circ \mathit{metricDuplicatedFormulas}
\end{aligned}
$$

To help with the conversion of the value of the metric with its associated risk, an auxiliary function *risk* was created and, given the threshold values, converts the value of a metric to its corresponding risk. Note that we use the thresholds proposed and validated for regular spreadsheet formulas [28], since the formulas in ClassSheet models are defined in a similar way.

$$
\begin{aligned}
&\mathit{risk} :: \mathit{Int} \to \mathit{Int} \to \mathit{Int} \to \mathit{Int} \to \mathit{Risk}\\
&\mathit{risk}\ \mathit{low}\ \mathit{moderate}\ \mathit{high}\ n \mid\ & n < \mathit{low}\ &= \mathit{None}\\
&\qquad\qquad\qquad\qquad\quad \mid\ & n < \mathit{moderate}\ &= \mathit{Low}\\
&\qquad\qquad\qquad\qquad\quad \mid\ & n < \mathit{high}\ &= \mathit{Moderate}\\
&\qquad\qquad\qquad\qquad\quad \mid\ & \mathit{otherwise}\ &= \mathit{High}
\end{aligned}
$$

Each of the smell functions use a metric function. These functions receive a model and evaluate the metric for each cell containing an attribute, returning the list of cells and the number of times the corresponding problem occurs in it. This can be represented with $\mathit{Model} \to [(\mathit{Cell}, \mathit{Int})]$ as the type of the metric functions. For each result pair of cell and number of times the problem occurs, we apply the product of function *id* and *risk*. The function *id* will maintain the cell intact, and the risk will transform the number of the occurrence of the problem (for instance, number of operations) into the correct risk.

In the following sub-sections, we explain in more detail how we implement each of the metrics introduced before.

Multiple Operations Metric. To evaluate this metric, we iterate over all the items in the formula of each attribute and count all those representing operations, namely functions (*ExpFun*) and operators (*ExpUnoOp* and *ExpBinOp*):

$$
\begin{aligned}
&\mathit{metricMultipleOperations} = \mathit{countIf}\ \mathit{isOperation} \circ \mathit{modelAttributes}\\
&\quad \textbf{where}\ \mathit{isOperation}\ (\mathit{ExpFun}\ \_\ \_) &&= \mathit{True}\\
&\qquad\quad \mathit{isOperation}\ (\mathit{ExpUnoOp}\ \_\ \_) &&= \mathit{True}\\
&\qquad\quad \mathit{isOperation}\ (\mathit{ExpBinOp}\ \_\ \_\ \_) &&= \mathit{True}\\
&\qquad\quad \mathit{isOperation}\ \_ &&= \mathit{False}
\end{aligned}
$$

Note that we show here a simplified version of this function as the implementation requires certain details not useful to understand our work.

Multiple References Metric. To evaluate the *multiple references* metric, we count the number of references present in the formula:

$$
\begin{aligned}
&\mathit{metricMultipleReferences} = \mathit{countIf}\ \mathit{isReference} \circ \mathit{modelAttributes}\\
&\quad \textbf{where}\ \mathit{isReference}\ (\mathit{ExpRef}\ \_\ \_) = \mathit{True}\\
&\qquad\quad \mathit{isReference}\ \_ = \mathit{False}
\end{aligned}
$$

Condition Complexity Metric. To evaluate the *condition complexity* metric, we count the number of *IF*s in the formula. Its implementation is very similar to the previous metrics and thus we do not show it here. Note that an *IF* is a function and counts as an operation for the *multiple operations* metric.

Long Calculation Chain Metric. To evaluate thi metric, a dependency tree is generated for each attribute and then its height is calculated:

$$
\begin{aligned}
&\mathit{metricLongCalculationChain}\ m = \mathit{map}\ (\pi_2 \triangle \mathit{chainLength})\ \mathit{attrs}\\
&\quad \textbf{where}\ \mathit{chainLength} = \mathit{treeHeight} \circ \mathit{evalDeps}\ [\,]\\
&\qquad \mathit{attrs} = \mathit{map}\ (((\mathit{getCellClassName}\ (\mathit{classes}\ m) \circ \pi_1) \triangle \mathit{cellName} \circ \pi_2) \triangle \pi_2)\\
&\qquad\qquad\quad (\mathit{modelAttributes}'\ m)\\
&\qquad \mathit{evalDeps}\ l\ \mathit{attr} = \mathit{Node}\ \mathit{attr}\ \mathit{deps}\\
&\qquad\quad \textbf{where}\ \mathit{deps} = \mathit{map}\ (\lambda r \to \mathit{evalDeps}\ ((\pi_1\ \mathit{attr}) : l)\ (r, \mathit{getAtt}\ r)))\ \mathit{refs}\\
&\qquad\qquad\quad \mathit{refs}\ = \mathit{filter}\ (\lambda r \to \neg\ (r \in l))\ (\mathit{references}\ (\pi_2\ \mathit{attr}))\\
&\qquad \mathit{references}\ c = [(cn, an) \mid \mathit{ExpRef}\ cn\ an \leftarrow \mathit{universeBi}\ c]\\
&\qquad \mathit{getAtt} = \mathit{fromJust} \circ (\text{`}\mathit{lookup}\text{`}\mathit{attrs})
\end{aligned}
$$

Duplicated Formulas. To evaluate the *duplicated formula* metric, a list with the formula parts of each attribute is generated and compared to the lists of the other attributes. The number of attributes with a sub-formula match is returned.

$$
\begin{aligned}
&\mathit{metricDuplicatedFormulas}\ m = \mathit{map}\ (\pi_1 \triangle \mathit{countDuplicates})\ \mathit{attrs}\\
&\quad \textbf{where}\ \mathit{countDuplicates}\ = \mathit{length} \circ \mathit{filter}\ \mathit{id} \circ \mathit{findDuplicates}\\
&\qquad \mathit{findDuplicates}\ \mathit{attr} = [\mathit{isDup}\ \mathit{attr}\ \mathit{attr}' \mid \mathit{attr}' \leftarrow \mathit{attrs}, \mathit{attr}' \neq \mathit{attr}]\\
&\qquad \mathit{isDup}\ \mathit{attr}\ \mathit{attr}' = \mathit{or}\ (\mathit{map}\ (\lambda x \to x \in (\pi_2\ \mathit{attr}'))\ (\mathit{drop}\ 1\ (\pi_2\ \mathit{attr})))\\
&\qquad \mathit{attrs} = \mathit{map}\ (\mathit{id} \triangle \mathit{cellSubForm})\ (\mathit{modelAttributes}\ m)\\
&\qquad \mathit{cellSubForm}\ (\mathit{CellFormula}\ (\mathit{Formula}\ \_\ e)) =\\
&\qquad\qquad\qquad\qquad [e' \mid e' \leftarrow \mathit{universeBi}\ e, \neg\ (\mathit{isTerm}\ e')]\\
&\qquad\quad \textbf{where}\ \mathit{isTerm}\ (\mathit{ExpVal}\ \_) = \mathit{True}\\
&\qquad\qquad \mathit{isTerm}\ (\mathit{ExpRef}\ \_\ \_) = \mathit{True}\\
&\qquad\qquad \mathit{isTerm}\ \_ = \mathit{False}
\end{aligned}
$$

The Haskell code used to specify the metrics makes use of generic programming using Uniplate [30], which allows for a concise way to traverse of data structures.

We have shown how the detection of ClassSheet smells presented in Section 3 are concisely implemented in Haskell. We use the Haskell programming

language because is the implementation language of the model-driven spreadsheet framework MDSheet, where we will define the smell refactoring/elimination as ClassSheet evolution.

5.2 Refactoring of ClassSheet Formulas

The refactorings presented in Section 4 can involve the creation or removal of formulas from the cells, but all of them change the cell contents. The MDSheet environment does not take into account evolution of cell formula specifically. It focuses more one the layout of the spreadsheet and the correct referencing of cells using named attributes to provide a more user-friendly interface to end users. Nevertheless, our environment supports setting cell values, and changes to the layout and cell contents are needed in order to perform these refactorings.

The refactoring of formulas in MDSheet lies on top of its bidirectional transformation engine [6]. This engine is specified as a set of operations that can be performed on the spreadsheet models, another set that can be performed on the spreadsheet data, and a relationship between these two sets. The relation between operations on these two artifacts, model and data, describes the equivalent set of operations that is needed to be applied on the other artifact after the original artifact is evolved with the set of operations defined on it.

Model Evolution. In order to evolve spreadsheet models, a set of operations were defined on them, as expressed by the following data type:

```
data Op_M : Model → Model =
    addColumn_M   Where Index                          -- add a new column
  | delColumn_M          Index                         -- delete a column
  | addRow_M      Where Index                          -- add a new row
  | delRow_M             Index                         -- delete a row
  | setLabel_M    (Index, Index) Label                 -- set a label
  | setFormula_M  (Index, Index) Formula               -- set a formula
  | replicate_M   ClassName Direction Int Int          -- replicate a class
  | addClass_M    ClassName (Index, Index) (Index, Index)
                                                       -- add a static class
  | addClassExp_M ClassName Direction (Index, Index) (Index, Index)
                                                       -- add an expandable class
```

This set of operations are the ones to be applied any time that a refactoring needs to be applied to evolve formulas with smells. Using these operations, we get the automatic coevolution of the instance using the bidirectional environment of MDSheet, that is, user will get the spreadsheet data automatically evolved after removing the smells from the model.

Data Evolution. The refactoring of model formulas presented in this paper not only affects spreadsheet models, but they are also applied to their instances with the following operations used by the bidirectional transformation engine:

```
data Op_D : Data → Data =
    addColumn_D  Where Index                -- add a column
  | delColumn_D        Index                -- delete a column
  | addRow_D     Where Index                -- add a row
  | delRow_D           Index                -- delete a row
  | AddColumn_D  Where Index                -- add a column to all instances
  | DelColumn_D        Index                -- delete a column from all instances
  | AddRow_D     Where Index                -- add a row to all instances
  | DelRow_D           Index                -- delete a row from all instances
  | replicate_D  ClassName Direction Int Int  -- replicate a class
  | addInstance_D ClassName Direction Model   -- add a class instance
  | setLabel_D   (Index, Index) Label        -- set a label
  | setValue_D   (Index, Index) Value        -- set a cell value
  | SetLabel_D   (Index, Index) Label        -- set a label in all instances
  | SetValue_D   (Index, Index) Value        -- set a cell value in all instances
```

With this we guarantee that the improvements made to the model are also passed on to the their respective instances.

Model and Data Coevolution. One guarantee provided my MDSheet is the always conformance of the instances to their models. This is achieved by relating model operations to data ones, and vice versa. Whenever a model operation is performed, this operation is converted to a set of data ones that perform the necessary evolution steps in the instance so that the conformity to the model is restored. The inverse is also available, that is, changes to the instances also automatically coevolve the model so both the instance and the model are always synchronized.

Refactoring by Evolution. There are multiple evolution steps that can be performed to refactor the formula of an attribute. For all refactorings, one of the operations to perform is to refactor the formula which is done as defined before.

For the *extract subformula* refactoring, a new attribute is needed to store the extracted subformula. This implies the use of a $setFormula_M$ to create the new attribute and another $setFormula_M$ to update the old formula, after extracting the subformula. However, this can be impossible to realize if there is no place where to add the new attribute. Thus, a $addColumn_M$ or a $addRow_M$ may be necessary to allocate space for the new attribute.

For the *extract common subformula* refactoring, the evolution steps are the same as for the *extract subformula* refactoring. The difference is that multiple subformulas can be extracted with this refactoring.

For the *merge branches* refactoring, only a $setFormula_M$ is needed, updating the old formula with its refactored version.

For the *merge formulas* refactoring, a $setFormula_M$ has to be applied to set the formula with the other merged formulas. Moreover, for each of the merged formulas, a $setLabel_M$ or $setFormula_M$ can be applied to remove the formulas that were merged. If any row or column became empty, functions $delRow_M$ or $delColumn_M$ can be performed to remove them.

5.3 MDSheet with Smell Elimination

After detecting the spreadsheet smells, the system will then provide a set of refactorings which can be applied to eliminate each smell. The user may choose to apply such refactoring(s) to the model so it no longer contains smells. If we look back at the smells detected in Figure 3, we can apply our refactorings and end up with the table shown on the right in Figure 4.

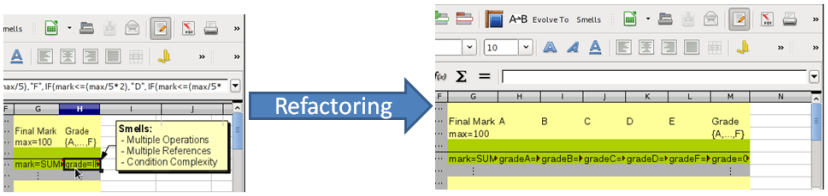

Fig. 4. Smell elimination in MDSheet.

Note that the data will be automatically coevolved so that it always maintains the instance's conformity to the model. This smell elimination on the model also eliminates smells in the data.

6 Related Work

There are several works that have focused on techniques based on smells detection to help improve the overall quality of spreadsheets.

One example of such work is presented in [28]. Here, the authors point a set of smells on formulas, define different threshold for these smells based on how frequent they appear on typical spreadsheets and suggest refactorings that improve the spreadsheet readability and maintainability. This is the work we adapt in this paper. A similar approach is taken by the same authors in [27], but this time they introduce smells that can point dangerous relations between worksheets in the same spreadsheet.

In [2], the authors created a extension (*RefBook*) for Microsoft Excel that detects and refactors a set of smells that find parts of the spreadsheet that contain, for example, unnecessary complexity and duplicated expressions. In this work, three empirical studies are performed to evaluate the capacity of this tool in improving the quality of spreadsheets and improving their readability, and the authors conclude that users prefer the improved quality of refactored sheets. There has also been tools built with the specific purpose of pointing and quantifying cells in a spreadsheet that can lead to potential problems. In [8], the authors use a smells-based technique, together with strategies that detect chains of inter-dependent cells and create new worksheets with color-based warnings.

In [26] the authors developed an extension for Excel, called *BumbleBee*, that is capable of detecting repetition of formulas in different zones of a spreadsheet and transforms them through a set of strategies. The authors conclude that more

than 70% of spreadsheets have a potential for the application of these transformations and that users are more capable of performing changes to spreadsheets using *BumbleBee*.

Another interesting work is presented in [7], where the authors suggest a new catalog of smells that can be used to further improve existing techniques such as the ones described in this section. Smells in Models is not an area as widely researched as smells in typical software programs. Of relevant reference however is [1]. Here, the authors propose a model-based quality assurance process that uses techniques that perform model quality analysis and model smells detection. Similarly to our approach, they also implement a tool that analysis and refactors models based on the Eclipse Modeling Framework.

7 Conclusion

The detection of code smells is a widely recognized software engineering technique that contributes to assessing the overall quality of a code repository. Indeed, identifying such bad design practices has now already been successfully explored in different contexts other than just source code, and namely in the context of spreadsheet engineering.

In the context of spreadsheets, however, smells have only been tackled at the level of concrete spreadsheets. This leaves out reasoning in the same way about spreadsheet abstract models.

This paper closes precisely this gap: we propose a catalog of smells for spreadsheet models, exploiting and adapting the catalog that has been proposed for spreadsheet instances. Finally, we follow the standard approach of associating refactorings with smells, in such a way that if these refactorings are adopted the identified smells disappear.

Acknowledgments. We would like to thank Jorge Mendes, Jácome Cunha, and João Saraiva for the help incorporating the ClassSheet smells in the MDSheet framework.

This work is part funded by ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT - Fundação para a Ciência e a Tecnologia within projects `FCOMP-01-0124-FEDER-022701`and Network Sensing for Critical Systems Monitoring (NORTE-01-0124-FEDER-000058), ref. BIM-2013_BestCase_RL3.2_UMINHO. The authors were funded by FCT grants `BIM-2013_BestCase_RL3.2_UMINHO`, `BI3-2013PTDC/EIA-CCO/116796/2010`, respectively.

References

1. Arendt, T., Taentzer, G.: Integration of smells and refactorings within the eclipse modeling framework. In: Proceedings of the Fifth Workshop on Refactoring Tools. pp. 8–15. WRT '12, ACM, New York, NY, USA (2012)

2. Badame, S., Dig, D.: Refactoring meets spreadsheet formulas. In: Proceedings of the 2012 IEEE International Conference on Software Maintenance (ICSM). pp. 399–409. ICSM '12, IEEE Computer Society, Washington, DC, USA (2012)
3. Beckwith, L., Cunha, J., Fernandes, J.P., Saraiva, J.: End-users productivity in model-based spreadsheets: An empirical study. In: IS-EUD'11. pp. 282–288. LNCS, Springer Berlin Heidelberg (2011)
4. Belo, O., Cunha, J., Fernandes, J.P., Mendes, J., Pereira, R., Saraiva, J.: Querysheet: A bidirectional query environment for model-driven spreadsheets. In: VL/HCC. pp. 199–200 (2013)
5. Cunha, J., Erwig, M., Saraiva, J.: Automatically inferring classsheet models from spreadsheets. In: Proceedings of the 2010 IEEE Symposium on Visual Languages and Human-Centric Computing. VLHCC '10, IEEE Computer Society (2010)
6. Cunha, J., Fernandes, J.P., Mendes, J., Pacheco, H., Saraiva, J.: Bidirectional transformation of model-driven spreadsheets. In: Theory and Practice of Model Transformations. LNCS, vol. 7307, pp. 105–120. Springer (2012)
7. Cunha, J., Fernandes, J.P., Mendes, J., Hugo Ribeiro, J.S.: Towards a Catalog of Spreadsheet Smells. In: The 12th International Conference on Computational Science and Its Applications. ICCSA'12, vol. 7336, pp. 202–216. LNCS (2012)
8. Cunha, J., Fernandes, J.P., Mendes, J., Martins, P., Saraiva, J.: Smellsheet detective: A tool for detecting bad smells in spreadsheets. In: Proceedings of the 2012 IEEE Symposium on Visual Languages and Human-Centric Computing. pp. 243–244. VLHCC '12, IEEE Computer Society, Washington, DC, USA (2012)
9. Cunha, J., Fernandes, J.P., Mendes, J., Pereira, R., Saraiva, J.: Querying model-driven spreadsheets. In: 2013 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). pp. 83–86 (2013)
10. Cunha, J., Fernandes, J.P., Mendes, J., Saraiva, J.: Embedding and evolution of spreadsheet models in spreadsheet systems. In: 2011 IEEE Symposium on Visual Languages and Human-Centric Computing. pp. 186–201. VLHCC '11 (2011)
11. Cunha, J., Fernandes, J.P., Mendes, J., Saraiva, J.: A bidirectional model-driven spreadsheet environment. In: 34rd International Conference on Software Engineering. pp. 1443–1444. ICSE '12 (June 2012)
12. Cunha, J., Fernandes, J.P., Mendes, J., Saraiva, J.: Extension and implementation of classsheet models. In: 2012 IEEE Symposium on Visual Languages and Human-Centric Computing. pp. 19–22. VLHCC '12 (2012)
13. Cunha, J., Fernandes, J.P., Mendes, J., Saraiva, J.: MDSheet: A Framework for Model-driven Spreadsheet Engineering. In: Proceedings of the 34rd International Conference on Software Engineering. pp. 1412–1415. ICSE '12, ACM (2012)
14. Cunha, J., Fernandes, J.P., Mendes, J., Saraiva, J.: Towards an evaluation of bidirectional model-driven spreadsheets. In: User evaluation for Software Engineering Researchers. pp. 25–28. USER' 12, ACM Digital Library (2012)
15. Cunha, J., Fernandes, J.P., Mendes, J., Saraiva, J.: Complexity Metrics for Classsheet Models. In: ICCSA (2). LNCS, vol. 7972, pp. 459–474. Springer (2013)
16. Cunha, J., Fernandes, J.P., Peixoto, C., Saraiva, J.: A quality model for spreadsheets. In: 8th Int. Conf. on the Quality of Information and Communications Technology, Quality in ICT Evolution Track. pp. 231–236. QUATIC '12 (2012)
17. Cunha, J., Fernandes, J.P., Pereira, R., Saraiva, J.: Graphical querying of model-driven spreadsheets. In: HCI'14. LNCS, Springer (2014), (to appear)
18. Cunha, J., Fernandes, J.P., Saraiva, J.: From Relational ClassSheets to UML+OCL. In: Proceedings of the Software Engineering Track at the 27th Annual ACM Symposium On Applied Computing. pp. 1151–1158. SAC '12, ACM (2012)

19. Cunha, J., Saraiva, J., Visser, J.: Discovery-based edit assistance for spreadsheets. In: 2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). pp. 233–237 (2009)
20. Cunha, J., Saraiva, J., Visser, J.: From spreadsheets to relational databases and back. In: Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation. pp. 179–188. PEPM '09, ACM (2009)
21. Cunha, J., Saraiva, J., Visser, J.: Model-based programming environments for spreadsheets. In: Programming Languages, LNCS, vol. 7554, pp. 117–133. Springer (2012)
22. Cunha, J., Visser, J., Alves, T., Saraiva, J.: Type-safe evolution of spreadsheets. In: Giannakopoulou, D., Orejas, F. (eds.) Fundamental Approaches to Software Engineering. Lecture Notes in Computer Science, vol. 6603. Springer (2011)
23. Engels, G., Erwig, M.: ClassSheets: automatic generation of spreadsheet applications from object-oriented specifications. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering. ACM (2005)
24. Erwig, M.: Software Engineering for Spreadsheets. IEEE Software 29(5) (2009)
25. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (Aug 1999)
26. Hermans, F., Dig, D.: Bumblebee: A transformation environment for spreadsheet formulas. Tech. rep., http://dx.doi.org/10.6084/m9.figshare.813347 (2013)
27. Hermans, F., Pinzger, M., van Deursen, A.: Detecting and visualizing inter-worksheet smells in spreadsheets. In: Glinz, M., Murphy, G.C., Pezzè, M. (eds.) ICSE. pp. 441–451. IEEE (2012)
28. Hermans, F., Pinzger, M., Deursen, A.: Detecting and refactoring code smells in spreadsheet formulas. Empirical Software Engineering pp. 1–27 (2014)
29. McCabe, T.J.: A complexity measure. IEEE Trans. Software Eng. 2(4) (1976)
30. Mitchell, N., Runciman, C.: Uniform boilerplate and list processing. In: ACM SIGPLAN Workshop on Haskell Workshop. pp. 49–60. Haskell '07, ACM (2007)
31. Nardi, B.A.: A Small Matter of Programming: Perspectives on End User Computing. MIT Press, Cambridge, MA, USA, 1st edn. (1993)
32. Panko, R.: Facing the problem of spreadsheet errors. Decision Line, 37(5) (2006)
33. Panko, R.: Spreadsheet errors: What we know. what we think we can do. Proceedings of the 2000 European Spreadsheet Risks Interest Group (EuSpRIG) (2000)
34. Pereira, R.: Querying for Model-Driven Spreadsheets. Master's thesis, University of Minho (2013)
35. Peyton Jones, S.: Haskell 98: Language and libraries. Journal of Functional Programming 13(1), 1–255 (2003)
36. Powell, S.G., Baker, K.R., Lawson, B.: A critical review of the literature on spreadsheet errors. Decision Support Systems 46(1), 128–138 (2008)
37. Rajalingham, K., Chadwick, D.R., Knight, B.: Classification of spreadsheet errors. In: Proceedings of the 2001 European Spreadsheet Risks Interest Group (EuSpRIG). Amsterdam (2001)