The Journal of Systems and Software 118 (2016) 234–250

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Evaluating refactorings for spreadsheet models

Jácome Cunha[a], João Paulo Fernandes[b], Pedro Martins[c], Jorge Mendes[c], Rui Pereira[c,*], João Saraiva[c]

[a] NOVA LINCS, DI, FCT, Universidade Nova de Lisboa, Portugal
[b] LISP - RELEASE, Universidade da Beira Interior & HASLab/INESC TEC, Portugal
[c] Universidade do Minho & HASLab/INESC TEC, Portugal

ARTICLE INFO

Article history:
Received 1 June 2015
Revised 18 December 2015
Accepted 18 April 2016
Available online 3 May 2016

Keywords:
Software refactoring
Model-driven engineering
Spreadsheets
Empirical study,

ABSTRACT

Software refactoring is a well-known technique that provides transformations on software artifacts with the aim of improving their overall quality.

We have previously proposed a catalog of refactorings for spreadsheet models expressed in the ClassSheets modeling language, which allows us to specify the business logic of a spreadsheet in an object-oriented fashion.

Reasoning about spreadsheets at the model level enhances a model-driven spreadsheet environment where a ClassSheet model and its conforming instance (spreadsheet data) automatically co-evolves after applying a refactoring at the model level. Research motivation was to improve the model and its conforming instance: the spreadsheet data.

In this paper we define such refactorings using previously proposed evolution steps for models and instances.

We also present an empirical study we designed and conducted in order to confirm our original intuition that these refactorings have a positive impact on end-user productivity, both in terms of effectiveness and efficiency.

The results are not only presented in terms of productivity changes between refactored and non-refactored scenarios, but also the overall user satisfaction, relevance, and experience.

In almost all cases the refactorings improved end-users productivity. Moreover, in most cases users were more engaged with the refactored version of the spreadsheets they worked with.

1. Introduction

Software refactoring (Fowler, 1999) is the process of modifying the source code of software programs without changing their semantics. That is to say that while improvements are expected on the non-functional attributes of a piece of software, it is mandatory that its associated functional attributes are not affected by refactorings.

Improvements can be achieved, for example, by transforming the software into a new version with reduced complexity, or with added expressiveness in either the code or its model (or both), or with diminished overall size (fewer methods, classes, or lines of code).

In practice, a significant set of automated refactorings is usually available for a concrete programming language. This reduces the overall programming effort, since due to the improved quality of refactored code traditional programming tasks become simpler and can be implemented faster (Fowler, 1999).

Because of its generic applicability, code refactoring has been studied in different contexts, ranging from software source code (Fowler, 1999; Mens and Tourwe, 2004) or software models (Einarsson and Neukirchen, 2012), to spreadsheets (Badame and Dig, 2012). We have ourselves proposed (Cunha et al., 2014) a series of refactorings for ClassSheets (Engels and Erwig, 2005).

ClassSheets are a high level, object-oriented modeling language for spreadsheets. Integrating concepts from the Unified Modeling Language (UML), this language provides a modular and abstract

* Corresponding author.
E-mail addresses: jacome@fct.unl.pt (J. Cunha), jpf@di.ubi.pt (J. Paulo Fernandes), prmartins@di.uminho.pt (P. Martins), jorgemendes@di.uminho.pt (J. Mendes), ruipereira@di.uminho.pt (R. Pereira), jas@di.uminho.pt (J. Saraiva).

http://dx.doi.org/10.1016/j.jss.2016.04.043
0164-1212/

methodology for dealing with spreadsheets, and namely to specify and maintain their business logic.

The appearance of this modeling language allowed us to develop an environment where concrete spreadsheets (or spreadsheet instances) are automatically derived from, and maintained together with abstract specifications (or spreadsheet models) (Engels and Erwig, 2005; Cunha et al., 2012c). This means that an evolution step on either the spreadsheet instance or its model is automatically propagated to the associated artifact, ensuring their consistency at all times (Cunha et al., 2012b).

In such a model-driven setting, we have shown (Cunha et al., 2015) that end-users are more efficient (that is, they complete equivalent tasks in less time) and effective (that is, they commit less errors) when they use a model-driven spreadsheet over regular spreadsheet data. In fact, that paper presents an empirical study showing that errors can be prevented by carefully reasoning about, and designing, a concise model, instead of doing so with a potentially large spreadsheet. This confirms an earlier idea that different, more refined, representations (or models, even though these representations are all at what we consider here the spreadsheet instance level) for data in a spreadsheet can improve productivity of end users (Beckwith et al., 2011).

In this paper we revise the refactorings proposed in Cunha et al. (2014) with the goal of improving the overall quality characteristics of ClassSheet models. The proposed refactorings are: *extract class, inline class, move attribute, move formula*, and *remove middle man*.[1] Moreover, we have specified these refactorings using our bidirectional transformational system previously introduced in Cunha et al. (2012b). This is the first contribution of this paper.

Later, we also assess in practice the refactorings catalog of (Cunha et al., 2014). With our work, we seek to find the answers to the following research questions:

(i) Do the spreadsheet instances (automatically) derived from refactored ClassSheet models allow end users to be more *efficient* than they would be if manipulating the instances derived from the corresponding original (non-refactored) models?
(ii) Do the spreadsheet instances derived from refactored ClassSheet models allow end users to be more *effective* than they would be if manipulating the instances derived from the corresponding original models?
(iii) Do spreadsheet users have a better experience working with the spreadsheet instances derived from refactored ClassSheet models instead of working with the instances derived from the corresponding original models?

That is to say that we propose to evaluate ClassSheet refactored models by analyzing the productivity in their instances, since these are standard spreadsheets, a software artifact that is used daily by millions of end users worldwide. We analyze the effectiveness and efficiency aspects of productivity, and also by measuring the overall experience of their users. With this goal in mind, we have designed and conducted an empirical study with spreadsheet end users, being this study, as well as the analysis of its results, the second and main contribution of this paper. In this study, we analyzed the quantitative and qualitative differences of the usage of (already) refactored spreadsheet models versus non-refactored spreadsheet models from an end-user's perspective.

The lessons learned from the results of our study are very promising. Through a series of statistical experiments, we found evidence that refactorings do allow improvements of either the efficiency or the effectiveness of its instances (or both), with the exception of a single refactoring from the refactoring catalog proposed in (Cunha et al., 2014).

Finally, we also gathered from the participants of our study feedback of a more qualitative nature. While the analysis of such feedback is exposed to a certain degree of subjectiveness, we believe that most of it provides further evidence that the refactorings of Cunha et al. (2014) allow the improvement of other spreadsheet characteristics such as readability, understandability or overall user satisfaction.

This paper is organized as follows. We start by introducing in Section 2 what model-driven spreadsheets are. In Section 3 we revise the previously introduced refactorings. In Section 4 we present the empirical validation of the refactorings proposed. Related work is presented in Section 5 and conclusions and future work in Section 6.

2. Model-driven spreadsheets

Engels and Erwig (2005)introduced the language ClassSheet to leverage handling spreadsheets to a more conceptual level. In a model-driven setting, a ClassSheet is the model and the spreadsheet data the corresponding instance. Indeed ClassSheets are more abstract than spreadsheets themselves, smaller, and easier to reason about. This language, which has a textual and a visual/graphical representation, has been embedded in spreadsheets themselves (Cunha et al., 2011). In such embedding the visual representation was used. Fig. 1 shows an embedded ClassSheet model of a small warehouse for a bar/coffee shop distribution (the numbered areas will be referenced in Section 3). Note this spreadsheet does not represent the actual data as it is shown in a second spreadsheets in Fig. 2.

On the top half (rows 1 through 6), we have three classes: **Product, Client**, and **Order. Product** (cell range A3:B5 and J3:K5) contains a product ID, its Name, Unit Price, and amount in Stock, while expanding vertically (indicated by the ellipsis on row 5). **Client** (cell range C1:G2) contains the client's Name, along with his/her Address, City, and Country, and expands horizontally (indicated by the ellipsis on column I). The **Order** (cell range C3:H5) is a relationship class which arises due to the joining of a **Product** and a **Client**. This class contains a Quantity value of the product, an Order Date, a product Category, a Sold Price formula to calculate the price, and the warehouse's ToSellprice (expected price) for selling all of that product. The ID in the **Client** class references their **Contact Info**, a class on the bottom half (cell range F8:H11), which has the client's Telephone and Email. The Seller's ID in the **Order** class references the **Seller** class (cell range A8:B11) which references the **SellInf** class (cell range C8:E11) containing the Name, Cell number, and Home number of the seller. These last three classes expand vertically.

Fig. 2 illustrates an instance of the model from Fig. 1. Starting from the bottom left corner, in a counter-clockwise direction, we can see instances for the Seller, SellInf, ContactInf, Order, two instances of Client (with the names Tiago C. and Marco C.) and four instances of Product.

Using ClassSheets we have created MDSheet (Cunha et al., 2012c), a framework that provides a bidirectional model-driven spreadsheet environment. The techniques and language described in that work allow transformations/evolutions from models to be automatically applied to the corresponding instances and vice-versa, as illustrated in Fig. 3.

Given a spreadsheet conforming to a ClassSheet, the user can evolve the model through an operation of the set Op_M, or the instance through an operation of Op_D. The performed operation on the model/instance is then automatically transformed into the corresponding set of operations on the instance/model using the *to*

[1] As the names suggest, and since ClassSheets resemble the object-oriented (OO) paradigm, the refactorings we proposed are based on the ones available in the OO realm.

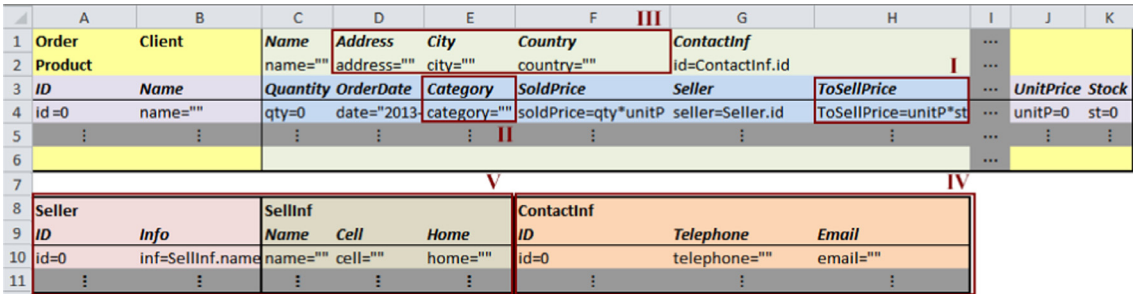


Fig. 1. An example of an embedded ClassSheet model for a warehouse goods distribution.

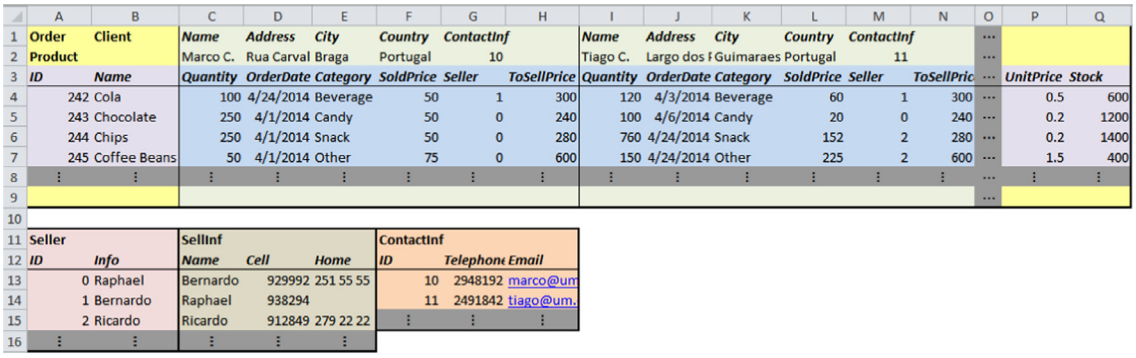


Fig. 2. An instance for the warehouse goods distribution ClassSheet shown in Fig. 1.

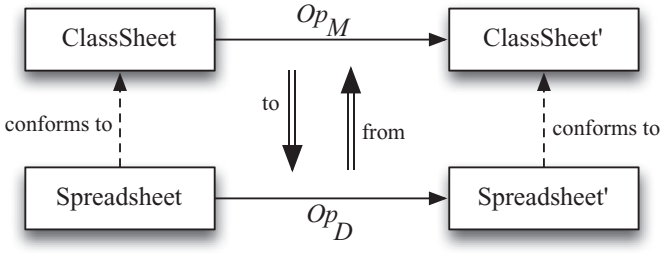


Fig. 3. Diagram of MDSheet bidirectional transformation system.

and *from* transformations, respectively. A new model and data is obtained with the new data conforming to the new model.

For transformations of models and instances, Op_M and Op_D respectively, we have defined a grammar that represents the functions operating on each one. To implement refactorings on models we will use the former, and benefit from the existing infrastructure: operations on models will automatically reflect themselves on updates on the instance.

ModelOperation defines the grammar for the Op_M operations. The application of an update op_M: Op_M to a model m: *Model* is denoted by $op_M\, m$: *Model*.

$$
\begin{array}{lll}
\textbf{data}\ \mathit{ModelOperation} : \mathit{Model} \to \mathit{Model} = & & \\
\quad \mathit{addColumn}_M & (\mathit{Where}, \mathit{Index}) & -\text{ add a new column} \\
\quad \mid \mathit{delColumn}_M & (\mathit{Index}) & -\text{ delete a column} \\
\quad \mid \mathit{addRow}_M & (\mathit{Where}, \mathit{Index}) & -\text{ add a new row} \\
\quad \mid \mathit{delRow}_M & (\mathit{Index}) & -\text{ delete a row} \\
\quad \mid \mathit{setLabel}_M & (\mathit{Point}, \mathit{Label}) & -\text{ set a label} \\
\quad \mid \mathit{setFormula}_M & (\mathit{Point}, \mathit{Formula}) & -\text{ set a formula} \\
\quad \mid \mathit{replicate}_M & (\mathit{ClassName}, \mathit{Direction}, \mathit{Int}, \mathit{Int}) & -\text{ replicate a class} \\
\quad \mid \mathit{addClass}_M & (\mathit{ClassName}, \mathit{Point}, \mathit{Point}) & -\text{ add a static class} \\
\quad \mid \mathit{addClassExp}_M & (\mathit{ClassName}, \mathit{Direction}, \mathit{Point}, \mathit{Point}) & -\text{ add an expandable class} \\
\quad \mid \mathit{delClass}_M & (\mathit{ClassName}) & -\text{ add a static class}
\end{array}
$$

The first operation, $addColumn_M$, adds a column in a particular place in the spreadsheet. The *Where* argument specifies the relative location (*Before* or *After*) and the given *Index* defines the position where to insert the new column. This solves ambiguous situations, like for example when inserting a column between two other columns from distinct classes. In an analogous way, the second operation, $delColumn_M$, deletes a given column of the spreadsheet. The operations $addRow_D$ and $delRow_M$ behave as $addColumn_M$ and $delColumn_M$, but work on rows instead of on columns.

The operation $setLabel_M$ allows to set a new label (*Label*) to a given cell, that is, it defines the value of a cell which cannot be an attribute, only text or numerical values. The cell position is given by *Point*, which represents the indexes of the column and of the row of such cell (Index, Index).

Other operations include $setFormula_M$ which allows to define a formula, *Formula*, on a particular cell. On the model side, a formula may be represented by an empty cell, by a default plain value (for instance, an integer or a date) or by a function application.

The operation $replicate_M$ allows to replicate (or duplicate) a class. The following two operations allow the addition of a new class to a model: $addClass_M$ adds a new static (non-expandable) class named *ClassName*, and $addClassExp_M$ creates a new expandable class. The *Direction* parameter specifies if it expands horizontally or vertically. Finally, the function $delClass_M$ is used to delete a class and respective cells.

Finally, the $delClass_M$ deletes a given class.

Note that MDSheet enforces that each evolution step on the model or on the instance immediately and automatically affects the other artifact. This means that both the instance and the model are kept synchronous at all times. For more information on the MDSheet system please refer to Cunha et al. (2012b, 2012c, 2015).

In the next sections we will define a set of refactoring for ClassSheets that are based on this existing framework.

3. Model-driven spreadsheets refactoring

As we are building our refactoring system on top of a model-driven framework, we can and will use some of its features. Indeed, to express the refactorings we will introduce a set of auxiliary functions defined using the already existing model evolution steps presented in the previous section. Thus, each refactoring will be expressed as a set of transformations which can be mapped

into evolution steps. Recall that when these steps are applied to the model, they also automatically make the instances co-evolve accordingly. This will ensure that any refactoring applied to the model will always produce the corresponding correct spreadsheet instance.

3.1. Refactorings as evolution steps

The refactorings are defined using a set of auxiliary functions defined next in *ModelRefactoring*. Such functions return an ordered list of the operations (model evolution steps) that must be applied to the models to refactor them.

```
data ModelRefactoring : Spreadsheet → [ModelOperation] =
    AddShiftForm    (Line, Point, AttributeName, Value)      -- add a formula
  | AddShiftAtt     (Line, Point, AttributeName, Value)      -- add an attribute
  | AddShiftRef     (Line, Point, AttributeName, Value)      -- add a reference
  | DeleteShift     (ClassName, AttributeName)               -- delete a cell
  | CreateClass     (ClassName, Direction, Point, Point)     -- create a new class class
  | DeleteClassShift (ClassName)                             -- delete a classs
```

The *AddShiftForm* function adds a formula to the model, shifting other cells if necessary. It receives the indication argument if is to be represented by column or a row (*Line* is either *Column* or *Row*), where it is to be placed (*Point*), what is the name of the attribute that will contain the formula (*AttributeName*) and the formula itself (*Value*). Before inserting the formula, the function checks if the target cell is empty (using the function *isCellEmpty*), and adds a column or a row if it is not.

```
AddShiftForm : (Line, Point, AttributeName, Value) → [ModelOperation]
AddShiftForm (line, p, an, v) = [λm → setFormula_M (p, (CellFormula an v)) (shift m)]
  where shift m = if isCellEmpty p m then m
                  else (if line ≡ Column then addColumn_M (Before, p) m
                        else  addRow_M (Before, p) m)
```

The *AddShiftAtt* and *AddShiftRef* functions are similar to *AddShiftForm*, where the semantics of the values to be inserted change according to the function.

The *DeleteShift* function deletes an attribute (identified by its class name, *ClassName*, and its attribute name, *AttributeName*) from the model, and shifts cells if necessary. It deletes the attribute by setting the value of its cell to empty. If this operation results in an empty column or in an empty row, the function removes the empty column or empty row.

```
DeleteShift : (ClassName, AttributeName) → [ModelOperation]
DeleteShift (cn, an) = [setLabel_M ((x, y), (CellValue ""))
                       , λm → if isColumnEmpty x m then delColumn_M ((x, y)) m else m
                       , λm → if isRowEmpty y m then delRow_M ((x, y)) m else m]
  where (x, y) = getAttributePosition (cn, an)
```

The *CreateClass* function adds a new class to the model, allocating the required space. The *replicate* function used in *CreateClass* replicates a value a given number of times. In this way, it is possible to allocate the necessary columns or rows for the new class. The operator ++ concatenates two list.

```
CreateClass : (ClassName, Direction, Point, Point) → [ModelOperation]
CreateClass (cn, dir, (x0, y0), (x1, y1)) =
  replicate (cHeight, (λm → if width m ≡ cWidth then addRow_M (Before, y0) m else m)) ++
  replicate (cWidth, (λm → if height m ≡ cHeight then addColumn_M (Before, x0) m else m)) ++
  [if Direction ≡ None
    then addClass_M (cn, (x0, y0), (x1, y1))
    else  addClassExp_M (cn, dir, (x0, y0), (x1, y1))]
  where cWidth = x1 − x0 + 1
        cHeight = y1 − y0 + 1
```

The *DeleteClassShift* removes a class from the model. Only the class name is needed to complete this operation. The function $delClass_M$ already performs the removal/shifting of cells.

$$DeleteClassShift : (ClassName) \rightarrow [ModelOperation]$$
$$DeleteClassShift\ (cn) = [delClass_M\ (cn)]$$

All of our refactoring functions return the joining of the ordered lists from the output of our auxiliary functions. This concatenated list is used by MDSheet to evolve the ClassSheet models to their refactored version. However, we omit such joining to simplify the algorithms shown. To note, all the *shift* functions automatically organize and shift surrounding cells.

We will now present a set of refactorings for ClassSheets, how they apply to the models of Fig. 1, and how they can be implemented in MDSheet. For each of them, we discuss when and why they would be needed, how to refactor, and express the refactoring function.

3.2. Move formula

Feature envy

In the OO paradigm it may happen that a method is used by or uses too many attributes from another class. This undesirable phenomenon is called *feature envy* and has also been identified in spreadsheets (Hermans et al., 2012).

In our case, we can consider classes as being the ClassSheet classes and the methods as being their formulas. Indeed this method–formula analogy as also been used in Hermans et al. (2012).

Thus, we move formulas when they are more interested in and used by attributes of another class than the class on which they are defined.

If we closely analyze the `ToSellPrice` formula (shown in Fig. 1, with the red frame marked with an I), we can see that not only does it suffer from feature envy, but semantically it makes more sense being in the **Product** class since it is defined using attributes from that class and not from the class **Order** (where it is now defined).

Refactoring

Fowler typically suggests putting a method in the class which contains most of the data used by it (*move method attribute*). This too can be applied to model-driven spreadsheets. We can move the `ToSellPrice` formula from the **Order** class to the **Product** class. This can be seen in Fig. 4, in column L, since the formula has now the same background color as the other attributes in **Product** (namely **UnitPrice** and **Stock**).

This refactoring has the potential to improve the representation and understandability of the spreadsheet (Hermans et al., 2012; Conway and Ragsdale, 1997), as the formula is now closer to the attributes it uses, and semantically in the correct class.

Evolution

The steps shown in Refactoring 1 describe the move formula refactoring

To execute the move formula refactoring on Fig. 1 to obtain Fig. 4 we would run:

MoveFormula(Order, ToSellPrice, Product, L4, Column)

`MoveFormula` takes information from the class `Order`, namely the value `ToSellPrice`, and moves it to the class `Product`, to the position L4.

3.3. Move attribute

When/Why

Another common refactoring for model-driven spreadsheets is *move attribute*. A simple reason to use it would be moving an attribute in a class to visually enhance the readability, or move an attribute between classes due to information evolution.

Refactoring 1 MoveFormula

Require: fromClass, formulaName, toPoint, line
Ensure: [*ModelOperation*]
formula ← GetFormula(fromClass, formulaName)
DeleteShift(fromClass, formulaName)
AddShiftForm(line, toPoint, formulaName, formula)

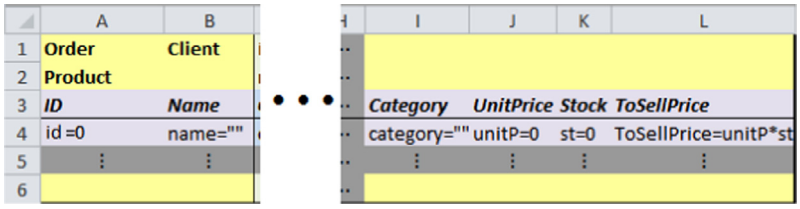


Fig. 4. Move formula and move attribute refactoring on ToSellPrice and Category respectively.

Another reason would be in a relationship class when it is detected that the instanced value of an attribute varies between one of the outer classes, and does not with the other. This means that the attribute might be in the wrong place, and should be placed in the class which directly affects the attribute. This problem can also be found in relational databases due to incorrect normalization (Maier, 1983; Cunha et al., 2009). We can see a sample of this occur in Fig. 2 - II on the `Category` attribute.

Refactoring

Here we choose the attribute we wish to change places, and choose what class and location in that class we want to change it to. Looking at Fig. 1 - II, we would move the `Category` attribute into the **Product** class, and obtain Fig. 4 as our new class.

Evolution

The steps listed in Refactoring 2 describe the move attribute refactoring.

To execute the move attribute refactoring on the model presented in Fig. 1 to obtain the one presented in Fig. 4 we would run:

MoveAttribute(Order, Category, Product, I4)

`MoveAttribute` moves the value `Category` from the class `Order` to the class `Product`, more precisely to the position `I4`.

3.4. Extract class

When/Why

Models can grow over time due to the creation of new attributes. This growth eventually causes the model to become too complicated and hard to understand. Where we once had a class with a clear purpose, we now may have a class doing the work of two. This is also a common scenario in the OO paradigm.

Since readability in a spreadsheet is important, as it is in any software, the moment we have a subset of information which is often times neglected, it might be a good practice to extract this subset, placing it aside as a new entity. For example, imagine that the users of our spreadsheet example do not tend to use the `Address`, `City`, and `Country` attributes, as shown in Fig. 1 - III. As these are a subset of client information, and make reading the **Client** class difficult, it is a good candidate for the extract class refactoring.

Refactoring

We first need to choose which subset of information we want to extract to a new class and create this new class with a new name. The previous attributes will be removed from the old class, and placed into the new class along with an `ID` attribute. Finally, the `ID` attribute is then referenced from the old class. This would be applied to produce Fig. 5 from Fig. 1 - III.

Refactoring 2 MoveAttribute

Require: fromClass, attributeName, toPoint, line
Ensure: [*ModelOperation*]
  attribute ← GetAttribute(fromClass, attributeName)
  DeleteShift(fromClass, attributeName)
  AddShiftAttr(line, toPoint, attributeName, attribute)

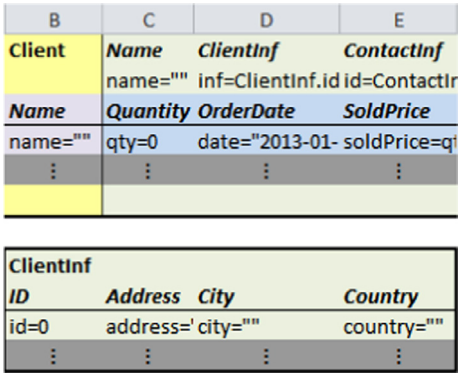


Fig. 5. Extract class refactoring on Address, City, and Country.

Evolution

The steps in Refactoring 3 describe the extract class refactoring.

To execute the extract class refactoring on the model in Fig. 1 to obtain the one in Fig. 5 we would run:

ExtractClass(Client, ClientInf, Vertical, B8, [address, city, country])

`ExtractClass` takes the class `Client` and creates the new class `ClientInf`. The new class grows vertically and starts on the position B8. The last argument is a list of values that will be extracted to the new class.

Refactoring 3 Extract Class

Require: fromClass, newClass, newClassPos0, newClassPos1, attrStartPoint, line, attrNames
Ensure: [*ModelOperation*]oPoint ← GetAttribute(fromClass, attrNames[0])
  expansion ← direction(fromClass)
  CreateClass(newClass, expansion, newClassPos0, newClassPos1)
  point ← attrStartPoint
  AddShiftAtt(line, point, 'id', '0')
  for all attrName ∈ attrNames **do**
    point ← nextPoint(point)
    attr ← GetAttribute(fromClass, attrName)
    DeleteShift(fromClass, attrName)
    if *value* ≡ *formula* **then**
      AddShiftForm(line, point, attrName, attr)
    else
      AddShiftAtt(line, point, attrName, attr)
    end if
  end for
  AddShiftRef(line, oPoint, newClass + 'Id', newClass + '.id')

Fig. 6. Inline class refactoring on the ContactInf class.

3.5. Inline class

When/Why

The inline class refactoring is the reverse of extract class. Inline class would be used in the cases where a class has insufficient justification of existing, due to not pulling its own weight, simply *not doing much*, or even having often consulted information. In these cases, one would remove the class, and join it with its outer-class or those which reference it.

Refactoring

When the user decides to use this refactoring, he/she would choose the unnecessary class to apply this refactoring to. The attributes which existed in this unnecessary class would be transferred over to the referencing classes, replacing the referencing `ID` attribute, and eliminating the class in question. We can see this refactoring applied to the model in Fig. 1 - IV to obtain the one in Fig. 6.

Evolution

The steps introduced in Refactoring 4 describe the inline class refactoring.

To execute the inline class refactoring on the model in Fig. 1 to obtain the one presented in Fig. 6 we would run `InlineClass`, which only has to receive as argument the name of the class `ContactInf`, as can be seen next:

InlineClass(ContactInf, *AllModelClasses*)

3.6. Remove middle-man

When/Why

A middle-man class is defined as a class which acts as a delegator between other classes. This class does not usually contain enough responsibility, logic, or purpose other than the simple delegation of operations/information. Along with being insufficiently useful, containing middle-mans usually complicates the structure and understanding of a spreadsheet (the same happens in the OO realm).

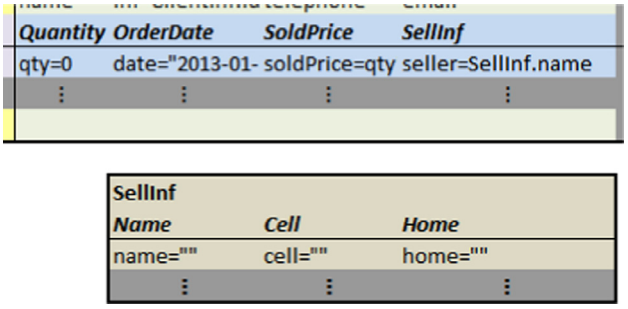


Fig. 7. Remove middle-man refactoring.

Refactoring

When a middle-man exists it should be removed. Furthermore, the classes which are being connected via the middle-man should be directly connected to each other.

Looking at Fig. 1 - V, we would remove the **Seller** class, which is only connecting itself to the **SellInf** class. We would then connect the **Order** class directly to the **SellInf** class, as shown in Fig. 7.

Evolution

Refactoring 5 list the steps which describe the remove middle-man refactoring.

To execute the remove middle-man refactoring on the model presented in Fig. 1 to obtain the one shown in Fig. 7 we would run:

RemoveMiddleMan(Seller, *AllModelClasses*)

3.7. Refactored example

Fig. 8 shows the complete refactored version of the ClassSheet we have been using as example. We were able to remove one useless class, and organize the data to be semantically correct. The refactorings also makes it easier for the user to read and use the spreadsheet more efficiently by joining attributes closer to their

Refactoring 4 Inline Class

```
Require: className, allClasses
Ensure: [ModelOperation]
  for all class ∈ allClasses do
    if REFERS(class, className) then
      refName ← GETREFERENCENAME(class, className)
      DELETESHIFT(class, refName)
      for all attr ∈ GETCLASSATTRIBUTES(className) do
        line ← GETLINEFOR(class)
        point ← GETLASTPOINTOF(class)
        attrName ← GETATTRIBUTENAME(attr)
        if attr ≡ formula then
          ADDSHIFTFORM(line, point, attrName, attr)
        else
          ADDSHIFTATT(line, point, attrName, attr)
        end if
      end for
    end if
  end for
  DELETECLASSSHIFT(className)
```

Refactoring 5 Remove Middle-Man

```
Require: className, allClasses
Ensure: [ModelOperation]
  for all referencingClass ∈ GETREFERENCESTO(allClasses, className) do
    refName ← GETREFERENCENAME(referencingClass, className)
    DELETESHIFT(referencingClass, refName)
    for all referencedClass ∈ GETREFERENCEDCLASSES(allClasses, className) do
      line ← GETLINEFOR(referencingClass)
      point ← GETLASTPOINTOF(referencingClass)
      ADDSHIFTREF(line, point, refName, referencedClass)
      for all attr ∈ GETCLASSATTRIBUTES(className) do
        attrName ← GETATTRIBUTENAME(attr)
        if attr ≡ formula then
          ADDSHIFTFORM(line, point, attrName, attr)
        else
          ADDSHIFTATT(line, point, attrName, attr)
        end if
      end for
    end for
  end for
  DELETESHIFTCLASS(className)
```

	A	B	C	D	E	F	G	H	I	J	K
1	Order	Client	Name	ClientInf	Telephone	Email	⋯				
2	Product		name=""	inf=ClientInf.id	telephone=""	email=""	⋯				
3	ID	Name	Quantity	OrderDate	SoldPrice	SellInf	⋯	Category	UnitPrice	Stock	ToSellPrice
4	id =0	name=""	qty=0	date="2013-01-	soldPrice=qty	seller=SellInf.name	⋯	category=""	unitP=0	st=0	ToSellPrice=unitP*st
5	⋮	⋮	⋮	⋮	⋮	⋮	⋯	⋮	⋮	⋮	⋮
6							⋯				
7											
8	ClientInf				SellInf						
9	ID	Address	City	Country	Name	Cell	Home				
10	id=0	address=""	city=""	country=""	name=""	cell=""	home=""				
11	⋮	⋮	⋮	⋮	⋮	⋮	⋮				

Fig. 8. The complete ClassSheet model after applying all the refactorings.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	Order	Client	Name	ClientInf	Telephone	Email	Name	ClientInf	Telephone	Email	⋯				
2	Product		Marco C.	0	2948192	marco@u	Tiago C.	1	2491842	tiago@un	⋯				
3	ID	Name	Quantity	OrderDate	SoldPrice	Seller	Quantity	OrderDate	SoldPrice	Seller	⋯	Category	UnitPrice	Stock	ToSellPrice
4	242	Cola	100	4/24/2014	50	Bernardo	120	4/3/2014	60	Bernardo	⋯	Beverage	0.5	600	300
5	243	Chocolate	250	4/1/2014	50	Raphael	100	4/6/2014	20	Raphael	⋯	Candy	0.2	1200	240
6	244	Chips	250	4/1/2014	50	Raphael	760	4/24/2014	152	Raphael	⋯	Snack	0.2	1400	280
7	245	Coffee Beans	50	4/1/2014	75	Raphael	150	4/24/2014	225	Ricardo	⋯	Other	1.5	400	600
8	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋯	⋮	⋮	⋮	⋮
9											⋯				
10															
11	ClientInf				SellInf										
12	ID	Address	City	Country	Name	Cell	Home								
13	0	Rua Carvalho	Braga	Portugal	Bernardo	929992	251 55 55								
14	1	Largo dos Pac	Guimaraes	Portugal	Raphael	938294									
15	⋮	⋮	⋮	⋮	Ricardo	912849	279 22 22								
16					⋮	⋮	⋮								

Fig. 9. The instance automatically co-refactored after refactoring the ClassSheet.

formulas, and placing often used attributes in classes easier to access (*e.g.,* joining the `Client`'s email and phone into the `Client` class). Fig. 9 shows the co-evolved instance, conforming to the model.

Comparing the original instance to the refactored one, we have 14 less data cells, a reduction of 15%, due to the elimination of redundant data. This reduction increases proportionally in relation to the data in the instance. For example, if we were to add one more client, we would have 22 less cells (17% reduction), and with two new clients 30 less cells (18% reduction). We can easily see that the larger our instance, the more impactful our refactorings.

In the next section we will present an empirical study showing that indeed the refactored models induce spreadsheet instance that are more effective and efficient to use from their users' perspective. In fact, we will also show these users have a better experience when using such improved spreadsheets.

4. Empirical validation

In the scope of software engineering research, empirically evaluating newly proposed techniques or methodologies is considered essential (Wohlin et al., 2012). Despite the fact that our work on model-driven spreadsheet refactoring had received positive feedback from the research community, its assessment in a real-world and practical setting is still crucial. For that reason, we have designed and executed an empirical study to evaluate our proposed model-driven spreadsheet refactorings.

In particular we are interested in evaluating the impact of the refactored spreadsheets. Although the refactorings we propose are for ClassSheet models, we intend to study their impact on the instances. This is so for several reasons:

- The creation and maintenance of spreadsheet models is important, but this does not tend to be done very often. Therefore, the maintenance of the spreadsheet instances/data is of much more importance and will happen much more often.
- One model can have several instances. Thus, the more impact we have on the instances, the better the spreadsheets can be considered as they have more impact, that is, they impact all the instances users, and not only the one model creator/manager. Thus, although we could evaluate the management of models themselves, to evaluate the impact of the refactorings on the instances is of much more importance.
- The users who create and manage the models are more advanced user than the ones operating the instances, that is, the spreadsheets themselves. Thus, it is more important to improve the working conditions of less advanced users as they may have more difficulties using bad spreadsheets that more advanced users (as the ones operating the models).

With this study we try to understand if our refactorings do in fact improve model-driven spreadsheets in terms of productivity and general user experience.

To analyze the productivity we designed a set of tasks for users to perform on refactored and non-refactored versions on the same scenario, measuring the amount of errors they committed each time, and also how much time it took them to perform such tasks.

To evaluate the users' experience, we asked a few questions about their preferences (which we will detail in the next subsections).

The perspective of this experiment is from the point of view of a researcher who would like to know whether or not there is a systematic difference between using non-refactored and refactored models.

This section will detail the different stages we underwent: design and preparation (Section 4.1), execution (Section 4.2), analysis (Section 4.3), interpretation (Section 4.4), and discussion (Section 4.5). We can summarize the scope of this study as suggested in Wohlin et al. (2012) as follows:

Analyze *the spreadsheet usage*
for the purpose of *evaluation*

with respect to its *productivity* and *user satisfaction*,
from the point of view of the *researcher*
in the context of *the usage of two different spreadsheets by Master students.*

4.1. Design

The goal of this study was to measure the quality of our proposed refactorings from a user perspective, and evaluate the differences between refactored and non-refactored models.

Specifically, we wanted to evaluate the *effectiveness, efficiency*, and *user acceptance* of our proposed model-driven refactorings over non-refactored versions:

- *Effectiveness*. Considering how common it is to find errors in spreadsheets, one of the objectives of our approach was to evaluate the effectiveness of refactored and non-refactored instances of spreadsheet models.
- *Efficiency*. Considering how important it is to be able to quickly perform actions in spreadsheets, we studied the time it takes to complete each task. This should provide information regarding the efficiency (or lack of it) of spreadsheet model refactorings, and observe the existence of potential tradeoffs between effectiveness and efficiency.
- *User acceptance*. We might have cases where our proposed refactorings may drastically reduce the time needed to perform normal spreadsheet tasks, but in turn may make it difficult for the user to understand and use the spreadsheet (or the inverse). Due to this, and due to the importance of user experience, we also want to evaluate the user acceptance.

This study was designed to be conducted in a controlled environment. In order to achieve this, we decided to perform the study in an off-line academic setting. The participants in this study were university students attending a Master's program. As we were evaluating the outcome of our proposed refactorings, we did not introduce ClassSheet models to the participants, only the refactored and non-refactored instances of those models. Our participants were asked to solve the traditional tasks of looking up, updating, and introducing new information in spreadsheets which were closely adapted from real-world examples.

4.1.1. Hypotheses

As previously stated, we want to test if our proposed refactorings do affect effectiveness (reduced error rates), the efficiency (increased productivity), and user acceptance. Thus, we can informally state three hypotheses:

1. In order to perform a given set of tasks, users are less error prone on the proposed refactored versions, comparing to non-refactored ones.
2. In order to perform a given set of tasks, users spend less time when using our proposed refactored versions instead of non-refactored ones.
3. In order to perform a given set of tasks, users prefer, perceive, and find it easier to use the refactored versions over the non-refactored ones.

Formally, three hypotheses are being tested: H_T for the time that is needed to perform a given set of tasks, H_E for the error rate found and H_U for the user choice rate. They are respectively formulated as follows:

1. *Null hypothesis*, H_{T_0}: The time to perform a given set of tasks using our refactored versions is not less than that taken with non-refactored versions. $H_{T_0} : \mu_d \leqslant 0$, where μ_d is the expected mean of the time differences.
   Alternative hypothesis, $H_{T_1} : \mu_d > 0$, *i.e.*, the time to perform a given set of tasks using our refactored versions is less than with non-refactored versions.
   Measures needed: time taken to perform the tasks.
2. *Null hypothesis*, H_{E_0}: The error rate in spreadsheets when using our refactored versions is not smaller than with non-refactored. $H_{E_0} : \mu_d \leqslant 0$, where μ_d is the expected mean of the differences of the error rates.
   Alternative hypothesis, $H_{E_1} : \mu_d > 0$, *i.e.*, the error rate when using our refactored versions is smaller than with non-refactored versions.
   Measures needed: error rate for each spreadsheet.
3. *Null hypothesis*, H_{U_0}: The user choice rate in spreadsheets when using our refactored versions is not smaller than with non-refactored. $H_{u_0} : \mu_d \leqslant 0$, where μ_d is the expected mean of the differences of the user choice rate.
   Alternative hypothesis, $H_{U_1} : \mu_d > 0$, *i.e.*, the user choice rate when using our refactored versions is smaller than with non-refactored versions.
   Measures needed: user's choice for each attribute (preference, understandability, ease of use, and correctness) for each spreadsheet.

4.1.2. Variables

The independent variables are: H_T the *time to perform the tasks*, for H_E the *error rate*, and H_U for the *user choice rate*.

4.1.3. Subjects and objects

The participants in this study were first year Master students, undergoing a course at the Universidade do Minho. A total of twenty students accepted our invitation and participated in our study. More details about the subjects participating in the study are presented in Section 4.3.

The objects for this study consisted of three different spreadsheets. Two of these (each with a specific scenario) were for the actual study, which are further described in Section 4.1.5. The third spreadsheet was used to perform a tutorial alongside the participants before beginning the study using the other two spreadsheets. This design choice attempts to minimize the threats to construct validity, namely the mono-operation bias (refer to Section 4.4.1 for more details on threats to validity).

4.1.4. Design

For this study, we followed a standard design with one factor and two treatments, as presented in Wohlin et al. (2012). The factor is *the spreadsheet usage*, that is, the insertion, update, and retrieval of information in a spreadsheet. The treatments are *refactored* and *non-refactored* spreadsheets. The dependent variables are measurable in a ratio scale, and hence a parametric test is suitable. However, as described further on, other conditions to apply a parametric test are not met. Thus, an equivalent non-parametric test is used.

Furthermore, blocking is provided in a way that each hypothesis is tested independently for each scenario. This reduces the impact of the differences between the two.

4.1.5. Instrumentation

As mentioned, our study was supported using three different spreadsheets, two for the actual study and one for a pre-study tutorial. We will begin by explaining the two used for the study. The refactorings are fully implemented in the MDSheet framework to automatically evolve the ClassSheets. The participants, however, did not have to interact with the framework, but only to visually look for information and fill in empty cells, as explained below.

The first has a scenario identical to the one used in our previous example (Section 2): a warehouse goods distribution spreadsheet. This spreadsheet had two worksheets of the warehouse example, with three of the five refactorings applied on one, and the remaining two on the other. In other words, we shuffled the five refactorings between the two worksheet copies. This spreadsheet scenario will be termed *orders* from now on.

In this first scenario, *orders*, we had seven instances of the client class, seven instances of client information, nine instances of the product class, 63 instances of the order class, and nine instances of seller class.

The second spreadsheet scenario was heavily based on a spreadsheet we obtained from a local food bank in Braga. This spreadsheet scenario contained information on the distribution of basic products and commodities. Specifically it contained information on: the products, the various institutions, and the amount of each product distributed to each institution. The product contained information on the product's name, code, amount in stock, category, and distribution unit. The institutions had information on its name, id, type, city, country, amount of lunch and dinner hampers, total hampers, website, email, and telephone. Replicating the structure, we created the ClassSheet classes for this scenario, giving us a total of six classes (product, institution, distribution, address, websites, and contact information). Once again, this spreadsheet had two worksheets of the same example, with three of the five refactorings applied on one, and the remaining two on the other. This spreadsheet scenario will now be termed *foodbank* from now on. The division of refactorings between the two worksheets allows us to evaluate the individual refactorings, and eliminate the tendecies of always selecting one spreadsheet (the fully refactored version) over the other (with no refactorings).

In our second scenario, *foodbank*, we had 10 instances of the product class, 18 instances of the institution class, 180 instances of the distribution class, five instances of the address class, and 18 instances for both the websites and contact information classes.

To note, the application of the refactorings did not alter the information represented in the spreadsheets, only the layout and structure of the various classes, sometimes adding or removing classes, but always maintaining consistency.

Guidelines were also provided to participants: they consisted of the list of tasks to be performed. For these tasks, we used Google forms, allowing us to present the tasks to the participants in their own computer in a simple way. Using this form, we had a page with the download links for the spreadsheets used in this study (both for the tutorial, and the two study scenarios), a pre-study questionnaire to obtain some basic profiling of our participants, simple instructions and comments, and a page to upload the spreadsheets post-study. The tasks were presented in the form, allowing us to take advantage of the integrated form mechanisms such as text-boxes and grids. The form was also divided into two parts, the first with the orders' tasks and the second with the foodbank's tasks, allowing them to focus on one scenario at a time.

For both scenarios, five tasks were presented. Each task is related to one of our five refactorings, allowing us to evaluate each refactoring individually and in an isolated form. Each refactoring has two associated tasks (one in each scenario). If in one scenario a data insertion/update task is given, a data retrieval task is given in the other scenario, and vice-versa. Having both types of tasks (read/write) gives us a better view of the refactor. Part of the tasks are shown (non-essential data is ommitted, being replaced by "[...]"):

Orders

(i) "Fill in the missing categories for the products. Consult the following table: [...]"
(ii) "How many different cities do we have clients in?"
(iii) "What is the ExpectedProficts value of: [...]"
(iv) "We need to contact our clients [...]. What are their telephone numbers?"
(v) "[...] takes care of all the [...] orders from now on. Update the spreadsheet to represent this information."

Foodbank

(i) "The category of [...]. Please update the spreadsheet to represent this information."
(ii) "We need to contact [...]. What are their emails?"
(iii) "Correct the total amount of meals [...] have"
(iv) "What are the distribution units of [...]?"
(v) "Fill in the missing institution addresses. Consult the following table: [...]"

Each task was to be completed in each worksheet, for both the refactored and non-refactored version. The tasks were shown twice, with a preceding "Using worksheet X", and each repetition had two text boxes for the participants to state the time they began and the time they finished the task (a large clock was displayed during the study for the participants).

In the tasks with data insertion/update, they would directly use the spreadsheets. While in data retrieval tasks they would answer the question using Google form's incorporated text boxes. In the case of the data retrieval tasks, each worksheet (from each spreadsheet) had different values so the answer would not be the same, eliminating learning effects and forcing the participants to actively retrieve data.

Additionally, at the end of each task, the participants were asked to choose between the first worksheet, the second, or neither on four attributes:

(i) "Which worksheet do you prefer to work with?"
(ii) "Which one do you feel is more understandable?"
(iii) "Which one do you feel was easier (to complete the task in)?'
(iv) "Which one do you believe has the more correct structure?"

Users also had a section where they could comment on the two worksheets, stating their opinions and what they would change.

Finally, we copied each of the two scenario spreadsheets, and inverse the worksheet order, providing yet another shuffle. In other words, in one copy we had the first worksheet as $worksheet_A$ and the second as $worksheet_B$, while in the second copy the first worksheet was $worksheet_B$ and the second $worksheet_A$. Now we have four physical spreadsheets (each scenario repeated) with shuffled worksheets. The two copied forms would be distributed between two randomly chosen groups (each form had the appropriate download links for the associated spreadsheets).

The third spreadsheet (tutorial spreadsheet) contains examples of the previous two scenarios to allow the participants to understand and work with these examples before beginning the study. This allowed them to familiarize themselves with the spreadsheet, the layout, and context. The tutorial consisted of insertion, update, and data retrieval tasks.

4.1.6. Data collection procedure

To correctly collect the spreadsheet data from the participants, and match the spreadsheet with the form, we would hand out, to each participant, a ticket with a number and the form url link before executing the study. Google's forms not only allowed us to create the tasks, but also allows an automatic form submission of the participant's responses. This information was automatically placed into a spreadsheet when the participant finished the study, and provided an upload link for the spreadsheets.

As Google forms automatically collected the form data, the steps were simple:

1. Handing out and filling study consentment form.
2. Handing out ticket number and study form url link.
3. Spreadsheet tutorial with participants.
4. Performing the study.
5. Submission of study form.
6. Submission of spreadsheets the participants edited during the study.
7. Collecting consentment forms.

All the participants were expected to perform all the tasks on the respective spreadsheets. We also asked the participants to complete the tasks in a quick but correct way. Thus, we did not prioritize answering time in relation to correctness. The goal of the study is not to compare one spreadsheet against another, but instead to compare the spreadsheet usage of our proposed refactored versions to the non-refactored ones.

4.1.7. Analysis procedure and evaluation of validity

The analysis of the collected data is achieved performing paired tests where the performance of each participant on the refactored version of the spreadsheet is tested against the non-refactored version. For this, the following tests are available: paired *t*-test, Wilcoxon sign rank test, and the dependent-samples sign-test.

Since the study is composed of several tasks, and each participant may not complete all of them, participants that do not complete all the tasks for both treatments of a spreadsheet will be discarded from the global analysis of that spreadsheet. This will allow us to compare the results of the remaining participants against each other, and not doing so could lead to incorrect illations given that a concrete task may be more error-prone than the others.

To ensure the validity of the data collected, several kinds of support were planned: constant availability to clarify any doubt, tutorial to teach and get used used to the spreadsheet, and supervise the work done by the subjects in a way that does not interfere with their tasks. This last point consists of navigating through the room and see which subjects look like they are having problems and try to help them if it is about something that does not influence the results of the study (problems using Google Forms, accessing and downloading the worksheets, for example).

4.2. Execution

This empirical study was performed in a classroom with twenty participants, all of them being Master students in informatics. All the participants performed the study at the same time, but the spreadsheets were shuffled in a way that no participant could use other participant's answers.

Initially, a consentement form was signed by the participants. This was to explain the purpose of the study, ensure them confidentiality, and inform them they could leave at anytime they wanted without any kind of reprisal. All the participants were encouraged to write whatever answer or opinion they thought best fit the questionnaire.

The concept of classheet instances was introduced, and some basic problems related to reading, analyzing and editing a spreadsheet were performed with them to ensure they had the basic necessary knowledge, and to ensure they were not completely oblivious to the tasks.

The study consisted of a form created with the aid of Google Forms, and spreadsheets the users were suppose to use during the study. All the participants were given a number which they used to download the spreadsheets to ensure different variations of the spreadsheets were spread across the participants and across the room.

The participants were also asked to use their own computers. This was to ensure they were working with their familiar hardware, and the time for the tasks was not spent unnecessarily on things like using an unfamiliar operative system or keyboards with unfamiliar layouts.

The first part of the form consisted on basic information related to the participants (gender, age) and their expertise with software models and spreadsheets. The other parts of the forms consisted on a set of tasks, and an upload site for the participants to upload the result of tasks that asked them to edit or insert information.

During the study the participants were supervised. This was to ensure that tasks that were not related to the study itself (download and upload spreadsheets, access the forms, problems interpreting the questions, for example) were rapidly answered and would not interfere with the time of the main tasks.

4.3. Analysis

The global analysis was performed with results from the participants who completed all the tasks for both scenarios, as described in Section 4.1.7. A initial step was to normalize all the shuffled scenarios and worksheets, after which we had a total of 19 out of 20 participants. Only 1 was fully discarded, because halfway through the study an external problem occurred preventing him from continuing.

We analyzed each refactoring from each scenario individually, i.e., we did not join the data from both scenarios into one. This allowed us to analyze the refactorings from two different uses: *read* and *write*. We then merged the data and analyzed the refactors from a global perspective.

4.3.1. Descriptive statistics

4.3.1.1. Subjects:. Basic information about the participants was gathered, namely their gender, age, background, and familiarity with spreadsheets and models (such as UML, ER, CPN, etc). From the twenty participants, nineteen were male and one was female. Most of them were aged between twenty-one and twenty-six, with two being over twenty-six. The subjects all came from an IT background, either *informatics engineering, computer science*, or *information technology and communication*. The average level of familiarity, from a scale of 1 (low) to 5 (high), for spreadsheets was 4, and for models was 3.

4.3.1.2. Time spent:. As expected, differences were found in the time that participants used to perform the tasks. The minimum times recorded on each scenario were by participants using our proposed refactorings. Table 1 shows the average time in seconds each participant took to achieve the given *read* tasks, both with and without our proposed refactorings.

Similar results were obtained for the *write* tasks, shown below in Table 2.

4.3.1.3. Error rates:. To evaluate the correctness of the spreadsheets produced during the study, error rates were used. Each task required either an answer to be given, or a series of cells to be filled and updated. In the cases where an answer would need to be given (*read* tasks), we considered a wrong answer as an error, so either 0% error or 100%. In the cases where the participant had to insert and update (*write* tasks), we considered every possible cell as a possible error. So if the participant had to insert information into 4 cells, and wrote 2 incorrect values, the error rate was 50%.

In Table 3, we show the amount of participants that made at least one error in the given tasks, both with and without our proposed refactorings, in both scenarios.

A superficial analysis shows that our refactored versions had less participants committing errors compared to the non-refactored version. The inline class tasks had no errors committed, the remove middle-man (read and write) and the move attribute

Table 1
Average and standard deviation time in seconds each participant took to achieve the *read* tasks.

	Move formula	Move attribute	Extract class	Inline class	Remove middle-man
Non-refact.	57.947 ± 45.227	53.474 ± 24.315	45.053 ± 19.293	49 ± 18.614	49.632 ± 20.699
Refactored	41.842 ± 28.775	30.526 ± 13.290	31.053 ± 23.350	48 ± 26.199	38 ± 13.561

Table 2
Average and standard deviation time in seconds each participant took to achieve the *write* tasks.

	Move formula	Move attribute	Extract class	Inline class	Remove middle-man
Non-refact.	58.421 ± 31.904	206.158 ± 108.106	98 ± 56.788	50.263 ± 40.395	82.579 ± 31.370
Refactored	34.842 ± 11.645	85.789 ± 56.983	86.421 ± 33.191	35.684 ± 35.657	86 ± 52.237

Table 3
Amount of participants that made at least one error.

	Move formula		Move attribute		Extract class		Inline class		Rem. middle-m.	
	Read	Write	Read	Write	Read	Write	Read	Write	Read	Write
Non-refact.	8	3	2	4	8	2	0	0	1	1
Refactored	1	1	2	0	3	0	0	0	0	0

Table 4
Wilcoxon p-values for comparison of time.

	Move formula	Move attribute	Extract class	Inline class	Remove middle-man
Read	0.01303000	0.00035440	0.01302000	0.35550000	0.02367000
Write	0.002135000	0.00010490	0.32160000	0.01110000	0.63620000

(read) tasks had same error rate, and the extract class (write) had low amount of errors overall.

4.3.1.4. Acceptance:. In most cases participants chose our proposed refactored versions over the non-refactored version in relation to the four attributes for user-acceptance: their preferred version, which version they felt was more understandable, which one they felt was easier (to complete the task in), and which one they believed had the more correct structure. The choices of the participants are represented in Fig. 10 and Fig. 11 for the *read* and *write* tasks respectively.

In Fig. 10, we can easily see that our refactored version was the most popular, albeit in Fig. 10(a) and (d) there does seem to be a small handful of participants who chose the non-refactored version in some of the attributes.

Looking at Fig. 11 we begin to see more of a competition. In Fig. 11(a) and (b), our version is the most popular. In Fig. 11(c), more participants chose the non-refactored version to be more understandable, while preferring the refactored version for the other three attributes. Finally, in Fig. 11(d) and (e), the results for the two versions seem to be very simillar, with a slight advantage in the non-refactored version.

4.3.2. Hypothesis testing

The significance level used throughout the evaluation of all the tests is 0.05. The evaluation of the tests was performed using the R environment for statistical computing (R Core Team, 2013).

4.3.2.1. Comparison of times. : The difference of times between the execution of tasks in the non-refactored and refactored versions do not follow a normal distribution. Thus, we used the Wilcoxon test, which is the best for these cases (Wohlin et al., 2012).

Looking at the results of applying the Wilcoxon test to test our hypotheses, shown in Table 4, we can see that the time taken to perform the tasks is statistically smaller using our refactorings as opposed to the non-refactorings in 7 out of 10 cases.

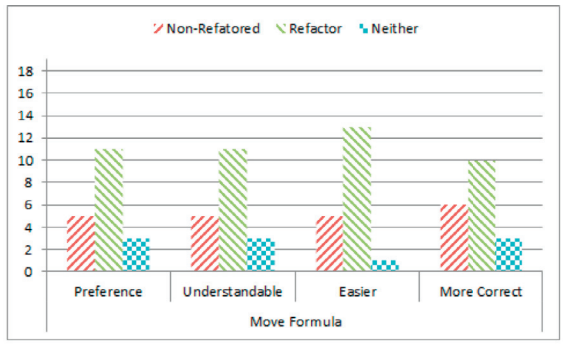
We decided to further analyze the three non-passing cases, extract class (write), inline class (read) and remove middle-man (write), and verify if the inverse hypothesis is true, in other words, are the non-refactored versions statistically faster. This produced Wilcoxon p-values of 0.6927, 0.6606, and 0.3802 respectively. Once again, the results are greater than the significance level, and in turn, we cannot conclude that one or the other version is statistically faster.

We took this one step further and merged the data from the tasks into one instead of separate read and write operations. This allowed us to analyze the refactorings from a global perspective of the refactorings and non-refactorings. In this case, we can see that the time taken to perform the tasks is statistically less using 4 out of our 5 proposed refactorings, as shown in Table 5.

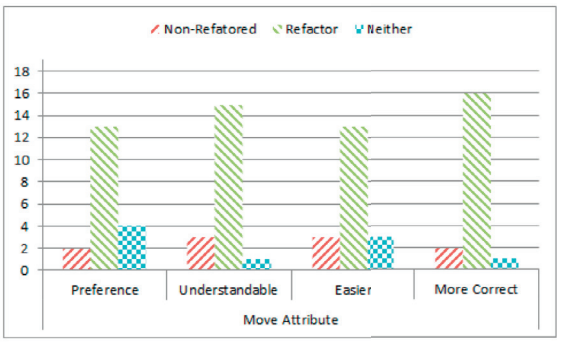
4.3.2.2. Comparison of error rates:. The differences of error rates between the execution of tasks in the non-refactored and refactored versions do not follow a normal distribution. Thus, we used once again the Wilcoxon test to test the null hypothesis for both versions in order to be able to compare the results.

The results obtained from the tests, shown in Table 6, show that the number of errors are statistically less using our refactored versions in 4 cases, move formula (read and write), extract class (read), and move attribute (write). In the case of extract class (write), we do not have enough statistical evidence to claim the same, albeit the only two errors which occurred were on the non-refactored version.

As expected and previously mentioned, the other cases did not produce any interesting results due to either having no errors, low amount of errors, or same error rate. Once again we merged the data from the tasks into one instead of separate read and write operations, and analyzed the error rates from a global perspective. We expected to have more interesting results as we also now have a bigger pool of error rates from both versions. The results are shown in Table 7. Here we can see that the errors rates are statistically smaller using 3 out of 5 of our refactorings. The inline class did not produce any results (as there were no errors), and the

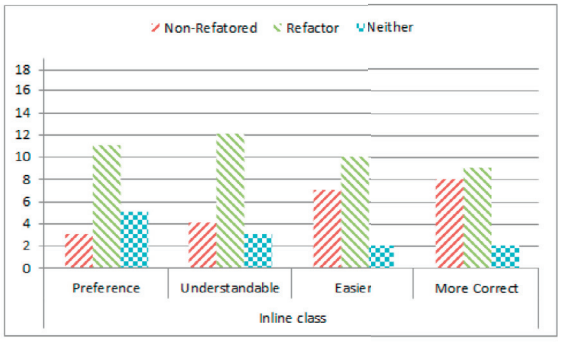


(a) Move Formula (read)

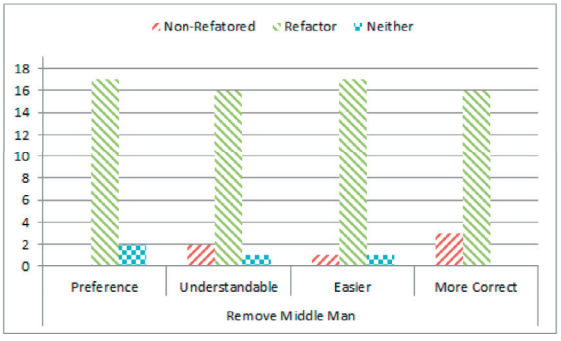


(b) Move Attribute (read)

(c) Extract Class (read)



(d) Inline Class (read)



(e) Remove Middle-Man (read)

Fig. 10. Post-task questions (read).

Table 5
Wilcoxon *p*-values for comparison of time from a global perspective.

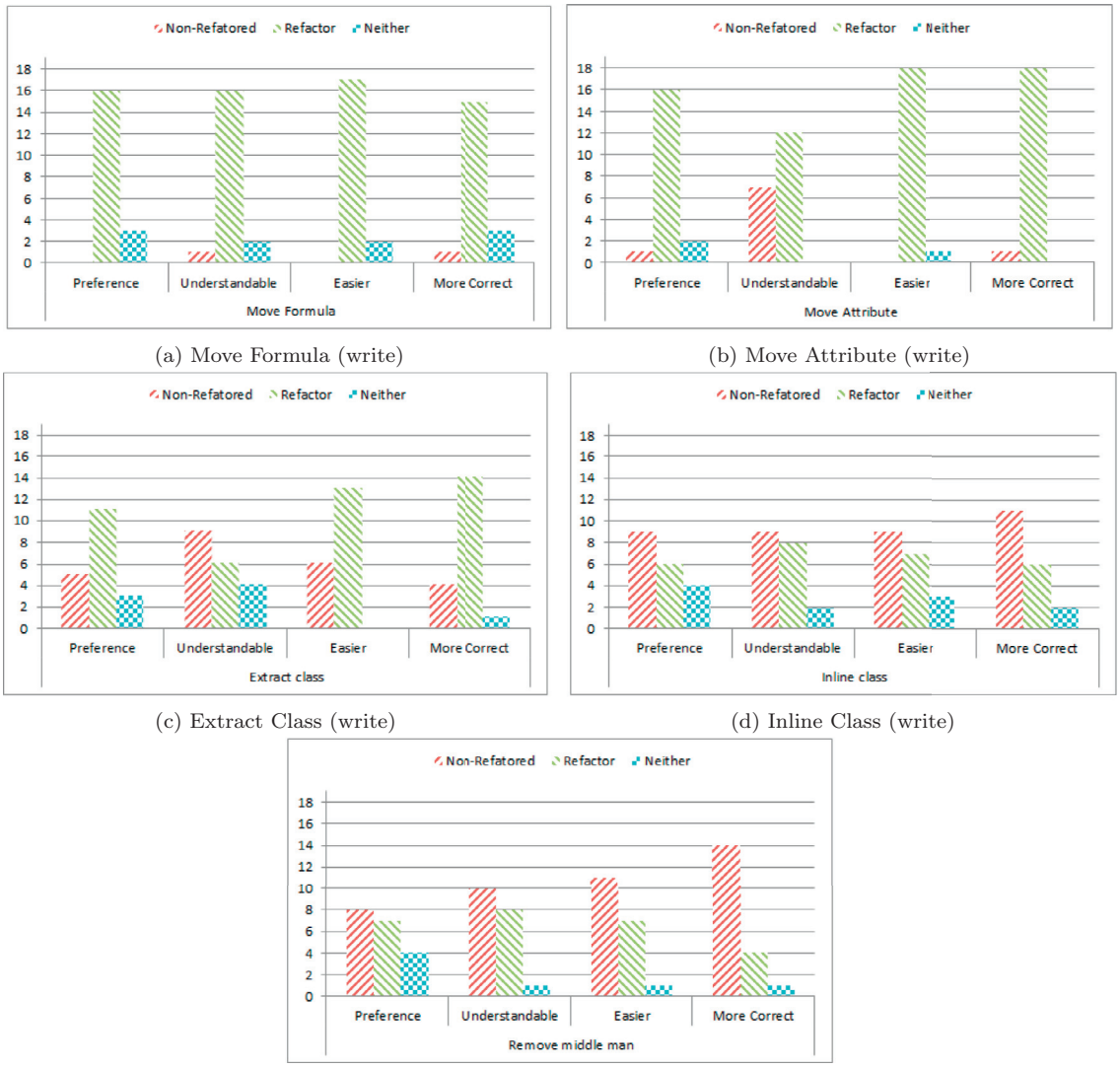
Move formula	Move attribute	Extract class	Inline class	Remove middle-man
0.0009704	0.000004768	0.03223	0.004707	0.5321

Table 6
Wilcoxon p-values for comparison of errors.

	Move formula	Move attribute	Extract class	Inline class	Remove middle-man
Read	0.03630000	0.00000000	0.03593000	0.00000000	0.50000000
Write	0.20710000	0.04860000	0.17290000	0.00000000	0.50000000

Table 7
Wilcoxon p-values for comparison of errors from a global perspective.

Move formula	Move attribute	Extract class	Inline class	Remove middle-man
0.01675	0.04876	0.01844	0.00000	0.1855



(a) Move Formula (write)

(b) Move Attribute (write)

(c) Extract Class (write)

(d) Inline Class (write)

(e) Remove Middle-Man (write)

Fig. 11. Post-task questions (write).

Table 8
Binomial sign test p-values for comparison of user choices of preference, understandability, ease of use, and correctness.

	Move formula		Move attribute		Extract class			Inline class		Rem. middle-man	
	R	W	R	W	R	W		R	W	R	W
P	0.2101	0.00003052	0.007385	0.0002747	0.00000763	0.2101	P	0.05737	0.6072	0.00001526	1.0
U	0.2101	0.00027470	0.007538	0.3593	0.0007286	0.6072	U	0.07681	1.0	0.001312	0.8145
E	0.09625	0.00001526	0.02127	0.00000763	0.00007629	0.1671	E	0.6291	0.8036	0.000145	0.4807
C	0.4545	0.0005188	0.001312	0.00007629	0.00000763	0.03088	C	1.0	0.3323	0.004425	0.03088

remove middle-man did not produce enough statistical evidence to pass our hypothesis.

4.3.2.3. User acceptance:. In order to test this hypothesis, we only considered the participants who chose one version over the other, and did not consider answers stating the user did not have a specific preference. For this, we used a binomial sign test, with the same 0.05 significance level, to test the null hypothesis for both versions.

Table 8 shows the p-values for our binomial sign tests. In more than half of the cases, we had enough statistical evidence to validate our hypothesis. In a majority of the read tasks, users chose our refactored version for the four attributes (preference (**P**), understandability (**U**), ease of use (**E**), and correctness (**C**)). Participants seem to be divided among the last three, extract class, inline class, and remove middle-man, for the write tasks. While two out of these three had enough statistical evidence showing that participants believed it to be more correct, we did not have enough evidence to validate our hypothesis for the other three attributes.

Finally, we once again merged the read and write tasks to form a global perspective analysis on our refactorings, which can be seen in Table 9. Our first three refactorings show statistical evidence to validate our hypothesis on our four attributes, with the last two yielding the same conclusion from the lower level analysis.

Table 9
Binomial sign test p-values for comparison of user choices of preference, understandability, ease of use, and correctness from a global perspective.

	Move formula	Move attribute	Extract class	Inline class	Remove middle-man
P	0.0001131	0.00000256	0.00003856	0.4583	0.007
U	0.0003241	0.007632	0.05011	0.2962	0.06525
E	0.00002236	0.00000077	0.0001162	1.0	0.06525
C	0.002102	0.00000012	0.00000194	0.6076	0.7428

Interestingly enough, if we look at Table 5, we can see how we do in fact improve the user's efficiency with the inline class during the write tasks. We also believe that both the extract class and inline class are heavily influenced by the way of how he/she prefers the data. In other words, if they rather work with more denormalized data (inline class) or more normalized data (extract class). This is also supported by the fact that users believed the extract class versions were more correct (in both read and write tasks), while the inline class versions were apparently not so clear. One possible reason is that these are MSc students have studied database normalization, and correct data design patterns, and possibly found it incorrect and harder to initially understand the idea of denormalizing data. Nevertheless, extract class and inline class are essentially opposite operations, and this allows users to structure their data to their preference.

4.4. Interpretation

The results from this empirical analysis suggest that the refactorings we propose create better instances, by increasing user productivity. However, some aspects have to be taken into account in this results.

4.4.1. Threats to validity

The goal of this study is to analyze if applying refactoring to a ClassSheet improves the overall quality of both the model and its instances, while ensuring that the setting presented represents the theory we developed with this work.

Next, we analyze possible threats to the validity of the results obtained, divided in the four categories of (Cook and Campbell, 1979):

4.4.1.1. Conclusion validity:. The low number of participants might imply a low statistical power from the results. To overcome this issue more powerful statistical tests were performed where possible, taking always into account the necessary assumptions.

Subjects were asked to perform the same task on refactored and non-refactored spreadsheets. We were carefully to shuffle the tasks so that half the participants started with the refactored version and half the participants started with the non-refactored spreadsheet, which means the overall results we not affected. However, there might exist some 'carry-over effect'.

4.4.1.2. Internal validity:. Several actions were taken with the objective of minimizing the effects of independent variables.

The time to perform the study was minimized as much as possible, to maintain the participants focused on the tasks, and the study was performed only once with these participants.

All the material for the study (forms, spreadsheets, the uploading website) were developed with the concerns of collecting only relevant information and minimizing distractions to the main tasks. Also, all the subjects whom we did not discard (19 out of 20) performed all the tasks (but in different order and with spreadsheets with different values but the exact same layout).

By controlling the group of participants and the tools they used we controlled and reduced threats to the internal validity of this study.

There is also the possibility of existing errors when measuring, for example, the times the participants took to perform each individual task. This was minimized by the fact that participants had the exact same background, performed the exact same tasks and were using familiar environments (all participants used their personal computers). This means variations in time are due to the tasks themselves, not to external factors, but introduces problems when generalizing (further below we explain why).

4.4.1.3. Construct validity:. To ensure construct validity on this study we defined several hypothesis that covered the aspects we wanted to analyze. Also, the participants were guaranteed not to be affected by the result or by their answers because they were not under evaluation (this was told to them several times).

The tasks we asked the participants to perform are typical spreadsheet tasks of information analysis, edition and retrieval, and they were introduced to these during a preliminar tutorial. We also gave the participants spreadsheets with different values (this was told to them), to avoid them copying the answers from other participants, since they were all in the same room.

We believe the construction of this study allows a correct evaluation of the implication of refactorings in the usage of spreadsheets.

4.4.1.4. External validity:. This validity is related to the ability of generalizing the results of the study to industrial practice. The different spreadsheets used were all based on spreadsheets we obtained from industrial partners, with minimal changes only to control the duration of the study. Nevertheless, since the group of participants was small and homogeneous, it is hard to generalize the results without further analyzing the domain in which the spreadsheets are being used.

4.4.2. Inferences

This study was performed in a very specific setting which makes it hard to generalize the results. The fact that a) the spreadsheets were based on real information provided by industrial partners and b) Master students can be comparable to professionals (Höst et al., 2000) which creates the possibility that applying refactorings to a spreadsheet model can be useful, by reducing potential errors and increasing the productivity of professionals.

4.5. Discussion

The results and analysis we present strongly suggest that refactorings have a positive impact on the usage of spreadsheets. These refactorings were inspired by the ones available for the OO paradigm, to which ClassSheets can be mapped (Cunha et al., 2012e).

Several researchers have proposed the detection of smells for spreadsheets (some also based on the OO realm) (Hermans et al., 2012; Pinzger et al., 2012; Cunha et al., 2012d; 2012a). Unfortunately, they usually do not propose the corresponding refactorings to eliminate the smells. Nevertheless, given the good results we achieved in this study it is expected that other refactorings may

also improve the quality of spreadsheets (although empirical validation is required). Thus, this study design can and should be used to evaluate other refactorings that may be proposed in the future.

Another interesting discussion includes the terms evaluated by our study. With the amount of errors being a common issue in spreadsheets, it is only natural to try to take that in account when considering techniques for spreadsheets engineering and usage.

One cannot evaluate the amount of errors without considering the time it takes to accomplish a certain task. If one proposes a technique that fully eliminates the errors but exponentially increases the amount of time it takes to do the job, then such technique probably will not be accepted by spreadsheet users, or any other users by that matter.

Finally, even improving users' efficiency and effectiveness, the new spreadsheet should engage users. Thus, it is crucial to evaluate the overall user experience and acceptance of our technique. As spreadsheets are naturally the end-users' language of choice, that is, the programming environment for non-professional programmers, their perspective is quite important.

Although we have shown that the refactorings we proposed can be used to improve spreadsheets, as they in general specify spreadsheet transformations, one may assume they may also be used to produce not so good spreadsheets. Indeed this is also the case for the refactorings available for other languages. Nevertheless. we characterized the context in which they can/should be used to improve spreadsheets.

5. Related work

After the popular book of Martin Fowler (Fowler, 1999) the concepts of code smells associated with program refactorings became widely used to improve the quality of software programs: code smells give an indication of poor quality of the software source code, in terms *e.g.* of its comprehensibility, while program refactorings are used to improve such perspective of quality. The main goal of program refactoring is to improve the overall quality of the software and not identify or to correct bugs/faults, nor to optimize/improve the performance of a program.

Although refactorings aim at improving software quality, several research studies show that they are not always effective in their aim (Stroggylos and Spinellis, 2007; Alshayeb, 2009). In this line, we have presented in this paper an empirical study that we conducted in order to confirm that the catalog of ClassSheet refactorings we proposed earlier does improve the overall quality of model-driven spreadsheet development.

Program refactorings have been widely applied/studied in different programming languages, like for example in C++ (Graf et al., 2007), Haskell (Li and Thompson, 2008), Erlang (Li and Thompson, 2012), and in software formalisms, like for example in software product line variability (Borba, 2011), in software security (Maruyama and Omori, 2011), in computer grammars (Kosar et al., 2004), and in UML models (Sunyé et al., 2001).

In the context of program refactorings, there are several empirical studies that validate their use by professional software engineers (Cunha et al., 2015; Kim et al., 2012), or the use of refactorings by software engineers in evolving software systems (Kim et al., 2011), or in inducing faults (Bavota et al., 2012), and in many other software development areas. The validation of refactorings in the context of (end-users) spreadsheet development has not received the same research work. This paper presents a first empirical study of end-users using refactorings for model-driven spreadsheets. Felienne's et al. paper (Hermans, Pinzger, van Deursen, 2014) presents a qualitative empirical study evaluating refactoring spreadsheet formulas by end-users. Our paper, however, presents a study where end-users have to perform more end-user oriented tasks, that is to say, more spreadsheets specific tasks, like for example, refactorings that involve layout operations.

There are also other works on applying program refactorings to spreadsheets formulas. Of special mention is Badame and Dig (2012), where the authors suggest a set of transformations to formulas using an Excel plugin. The fundamental difference to our work is that the refactorings we propose are applied to the spreadsheet model, which being more concise, makes the reasoning easier. WYSIWYG (Fisher II et al., 2002; Rothermel et al., 2000) is a tool for spreadsheet testing that helps users to find bugs and problems in spreadsheets. Contrary to our approach, this tool requires user input to find faults and works only individually on instances of spreadsheets.

Hermans *et al.* (Hermans et al., 2012; Pinzger et al., 2012) have various works on spreadsheet smells. They sometimes refer refactorings for the smells they introduce, but they specially focus on the detection of smells, not on their elimination. Their work is complementary to ours, as they focus on detecting spreadsheet problems and we, while sharing this same concern, have further focused in this work on solving some of them.

6. Conclusions

In this paper we have built on previous work where we proposed a set of refactorings for ClassSheet models, a formalism that has been widely used in the context of model-driven spreadsheet engineering.

While based on our experience in this context we have argued in the past that our refactorings provided *better* models, it is only after the work described in this paper that this argument has been fully realized and extended to their instances.

We have designed and conducted an empirical study comparing two scenarios: one where our refactorings are applied against one where they are absent. From the analysis of the obtained results, we have found statistical evidences that refactorings do allow spreadsheet engineering productivity improvements. These evidences are also substantiated with some qualitative data, of a more subjective interpretation, that we gathered from the participants of our study. Indeed, in most cases the study participants had a better experience working with the refactored versions.

We foresee several directions for future work, including a study on the applications of the refactorings and a user's experience with them. We tested variables that we consider of extreme importance for analyzing spreadsheet usage, namely effectiveness, efficiency, and user acceptance. The fact is that there is not a definition for spreadsheet quality, similar to the one described in ISO/IEC 9126 for software, where functionality, reliability or maintainability, for example, are clearly defined. To evaluate the quality of a spreadsheet several aspects could have been chosen, from the quality of the data itself, to the model complexity or readability. We are currently working on how well-known quality models can fit spreadsheets and how can quality be clearly defined. A first approach to this is described in Cunha et al. (2013).

We are also working on smells detection tools for spreadsheets, such as the one presented in Cunha et al. (2012a). This could be adapted to automatically identify when a refactoring could and should be applied.

Acknowledgements

We would like to thank Paulo Azevedo and Miguel Goulão for their wise advise on the analysis and presentation of the study results. We would also like to thank Pedro Henriques for his class time, and his students that volunteered to participate in our study.

This work is part funded by the 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 (Portuguese Foundation for Science and Technology) within project FCOMP-01-0124-FEDER-020484. The first, third and fifth author were funded by FCT: SFRH/BPD/73358/2010, Project NORTE-07-0124-FEDER-000062 and BI3-2013_PTDC/EIA-CCO/116796/2010_UMINHO, respectively.

References

Alshayeb, M., 2009. Empirical investigation of refactoring effect on software quality. Inf. Softw. Technol. 51 (9), 1319–1326.

Badame, S., Dig, D., 2012. Refactoring meets spreadsheet formulas. In: Proceedings of the 2012 IEEE International Conference on Software Maintenance (ICSM). IEEE Computer Society doi:10.1109/ICSM.2012.6405299.

Bavota, G., De Carluccio, B., De Lucia, A., Penta, M.D., Oliveto, R., Strollo, O., 2012. When does a refactoring induce bugs? an empirical study. In: Source Code Analysis and Manipulation (SCAM), 2012 IEEE 12th International Working Conference on. IEEE, pp. 104–113.

Beckwith, L., Cunha, J., Fernandes, J.P., Saraiva, J., 2011. End-users productivity in model-based spreadsheets: An empirical study. In: IS-EUD'11. Springer, Berlin Heidelberg, pp. 282–288.

Borba, P., 2011. An introduction to software product line refactoring. In: Proceedings of the 3rd International Summer School Conference on Generative and Transformational Techniques in Software Engineering III. Springer-Verlag, Berlin, Heidelberg, pp. 1–26.

Conway, D., Ragsdale, C., 1997. Modeling optimization problems in the unstructured world of spreadsheets. Omega 25 (3).

Cook, T., Campbell, D., 1979. Quasi-experimentation: design & analysis issues for field settings. Rand McNally College, Boston.

Cunha, J., Fernandes, J.P., Martins, P., Mendes, J., Saraiva, J., 2012a. Smellsheet detective: a tool for detecting bad smells in spreadsheets. In: Visual Languages and Human-Centric Computing. IEEE.

Cunha, J., Fernandes, J.P., Martins, P., Pereira, R., Saraiva, J., 2014. Refactoring meets model-driven spreadsheet evolution. In: Proceedings of the 9th International Conference on the Quality of Information and Communications Technology, Quality in Model Driven Engineering Track. To appear.

Cunha, J., Fernandes, J.P., Mendes, J., Pacheco, H., Saraiva, J., 2012b. Bidirectional transformation of model-driven spreadsheets. In: Hu, Z., de Lara, J. (Eds.), Theory and Practice of Model Transformations. Springer, Prague, pp. 105–120. http://dx.doi.org/10.1007/978-3-642-30476-7_7.

Cunha, J., Fernandes, J.P., Mendes, J., Saraiva, J., 2012c. MDSheet: a framework for model-driven spreadsheet engineering. In: Proc. of the International Conference on Software Engineering. IEEE.

Cunha, J., Fernandes, J.P., Mendes, J., Saraiva, J., 2013. Complexity metrics for spreadsheet models. In: et al., B.M. (Ed.), The 13th International Conference on Computational Science and Its Applications. LNCS, pp. 459–474. http://dx.doi.org/10.1007/978-3-642-39643-4_33.

Cunha, J., Fernandes, J.P., Mendes, J., Saraiva, J., 2015. Embedding, evolution, and validation of model-driven spreadsheets. IEEE Trans. Softw. Eng. 41 (3), 241–263. doi:10.1109/TSE.2014.2361141.

Cunha, J., Fernandes, J.P., Ribeiro, H., Saraiva, J., 2012d. Towards a catalog of spreadsheet smells. In: Proceedings of the 12th International Conference on Computational Science and Its Applications - Volume Part IV. Springer-Verlag, Berlin, Heidelberg, pp. 202–216. doi:10.1007/978-3-642-31128-4_15.

Cunha, J., Fernandes, J.P., Saraiva, J., 2012e. From relational ClassSheets to UML+OCL. In: Proceedings of the Software Engineering Track at the 27th Annual ACM Symposium On Applied Computing (SAC 2012). ACM, pp. 1151–1158.

Cunha, J., Mendes, J., Fernandes, J.P., Saraiva, J., 2011. Embedding and evolution of spreadsheet models in spreadsheet systems. In: VL/HCC'11. IEEE, pp. 186–201.

Cunha, J., Saraiva, J., Visser, J., 2009. From spreadsheets to relational databases and back. In: Proc. of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program manipulation.

Einarsson, H.T., Neukirchen, H., 2012. An approach and tool for synchronous refactoring of UML diagrams and models using model-to-model transformations. In: Proceeedings of the Fifth Workshop on Refactoring Tools. ACM doi:10.1145/2328876.2328879.

Engels, G., Erwig, M., 2005. ClassSheets: automatic generation of spreadsheet applications from object-oriented specifications. In: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering. ACM.

Fisher II, M., Cao, M., Rothermel, G., Brown, D., Cook, C., Burnett, M., 2002. Integrating Automated Test Case Generation into the WYSIWYT Spreadsheet Testing Methodology. Technical Report. Oregon State University, Corvallis, OR, USA.

Fowler, M., 1999. Refactoring: Improving the Design of Existing Code. Addison-Wesley.

Graf, E., Zgraggen, G., Sommerlad, P., 2007. Refactoring support for the C++ development tooling. In: Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion. ACM, New York, NY, USA, pp. 781–782. doi:10.1145/1297846.1297885.

Hermans, F., Pinzger, M., van Deursen, A., 2014. Detecting and refactoring code smells in spreadsheet formulas. Empir. Softw. Eng. 20 (2), 549–575.

Hermans, F., Pinzger, M., Deursen, A.V., 2012. Detecting and visualizing inter-worksheet smells in spreadsheets. In: Proceedings of the 2012 International Conference on Software Engineering. IEEE Press.

Höst, M., Regnell, B., Wohlin, C., 2000. Using students as subjects: a comparative study of students and professionals in lead-time impact assessment. Empir. Softw. Eng. 5 (3), 201–214. doi:10.1023/A:1026586415054.

Kim, M., Cai, D., Kim, S., 2011. An empirical investigation into the role of API-level refactorings during software evolution. In: Proceedings of the 33rd International Conference on Software Engineering. ACM, pp. 151–160.

Kim, M., Zimmermann, T., Nagappan, N., 2012. A field study of refactoring challenges and benefits. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. ACM, p. 50.

Kosar, T., Mernik, M., Zumer, V., 2004. Jart: grammar-based approach to refactoring. In: Proceedings of the 28th Annual International Computer Software and Applications Conference - Volume 01. IEEE Computer Society, Washington, DC, USA, pp. 502–507.

Li, H., Thompson, S., 2008. Tool support for refactoring functional programs. In: Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation. ACM, New York, NY, USA, pp. 199–203. doi:10.1145/1328408.1328437.

Li, H., Thompson, S., 2012. A domain-specific language for scripting refactorings in Erlang. In: de Lara, J., Zisman, A. (Eds.), Fundamental Approaches to Software Engineering. Springer Berlin Heidelberg, pp. 501–515. doi:10.1007/978-3-642-28872-2_34.

Maier, D., 1983. The Theory of Relational Databases. Computer Science Press.

Maruyama, K., Omori, T., 2011. A security-aware refactoring tool for java programs. In: Proceedings of the 4th Workshop on Refactoring Tools. ACM, New York, NY, USA, pp. 22–28. doi:10.1145/1984732.1984737.

Mens, T., Tourwe, T., 2004. A survey of software refactoring. IEEE Trans. Softw. Eng. 30 (2). http://doi.ieeecomputersociety.org/10.1109/TSE.2004.1265817.

Pinzger, M., Hermans, F., van Deursen, A., 2012. Detecting code smells in spreadsheet formulas. In: Proceedings of the 2012 IEEE International Conference on Software Maintenance. IEEE CS.

R Core Team, 2013. R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing. Vienna, Austria.

Rothermel, K.J., Cook, C.R., Burnett, M.M., Schonfeld, J., Green, T.R.G., Rothermel, G., 2000. WYSIWYT testing in the spreadsheet paradigm: an empirical evaluation. In: Proceedings of the 22nd International Conference on Software Engineering. ACM.

Stroggylos, K., Spinellis, D., 2007. Refactoring–does it improve software quality? In: Proceedings of the 5th International Workshop on Software Quality. IEEE Computer Society, Washington, DC, USA, pp. 10–. doi:10.1109/WOSQ.2007.11.

Sunyé, G., Pollet, D., Le Traon, Y., Jézéquel, J.-M., 2001. Refactoring UML models. In: Gogolla, M., Kobryn, C. (Eds.), UML 2001 — The Unified Modeling Language. Modeling Languages, Concepts, and Tools. Springer Berlin Heidelberg, pp. 134–148. doi:10.1007/3-540-45441-1_11.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., 2012. Experimentation in Software Engineering. Springer.

Jácome Cunha received the Licenciatura degree (five years) in mathematics and computer science from Universidade do Minho, Portugal, in 2006. He received the PhD degree in computer science from the same university in 2011. He joined the faculty of Universidade Nova de Lisboa, where he is currently an assistant professor. His main research interests include software engineering and programming languages, which he uses to improve software effectiveness, efficiency, and usability.

João Paulo Fernandes has graduated in mathematics and computer science from Universidade do Minho, Portugal, in 2004 (best of class). He received the PhD degree from the same university, following his work on the design, implementation and calculation of circular programs in 2009. In his research, he pursues rigorous ways to reason about programming, which he has successfully been able to apply in the context of functional programming, spreadsheets, language engineering and bidirectional transformations, and in the context of several research projects. Currently, he is an assistant professor at the Informatics Department, Universidade da Beira Interior, Portugal.

Pedro Martins received his MSc degree from Universidade do Minho in 2010, and a PhD in 2014 for his work on embedded attribute grammars. His research interests include the areas of, programming languages, functional programming, and software analysis. He is currently a Post-doc researcher at the University of California.

Jorge Mendes received his MSc degree from Universidade do Minho in 2012 for his work on evolution of model-driven spreadsheets. He is currently working toward the PhD degree in the MAP-i Programme at Universidade do Minho. He is a student member of the ACM. His research interests include the areas of software engineering, programming languages, functional programming, and model-driven engineering.

Rui Pereira is a computer science PhD student at the University of Minho, under the MAPi doctoral programme. He received his MSc degree in software engineering and business intelligence in 2013, with his thesis “Querying for Model-Driven Spreadsheets” under the SpreadSheets as a Programming Paradigm (SSaaPP) project. He continued working on his MSc work, extending it further into other approaches, along with working on spreadsheet model quality, human-computer interaction, and software analysis and transformation. He is a member of the bilateral project "Spreadsheet Model for the Real World", a collaboration between Portugal and Germany, funded by FCT, and the joint project “Towards Variational Software, Types, and Spreadsheets”, a collaboration between Portugal and the USA, funded by the FLAD foundation. He is currently working on his PhD in the area of Green Software Analysis with an awarded FCT grant.

João Saraiva received the MSc degree in computer science from the Universidade do Minho, Portugal, in 1993. He received the PhD degree in computer science from the University of Utrecht, The Netherlands, in 1999, and then joined the faculty at the Universidade do Minho where he is currently an assistant professor. His main research interests include programming languages, and software analysis, transformation and evolution.