Refactoring meets Model-Driven Spreadsheet Evolution

Jácome Cunha*†, João Paulo Fernandes*‡, Pedro Martins*, Rui Pereira*, and João Saraiva*
* HASLab/INESC TEC & Universidade do Minho, Portugal
† CIICESI, ESTGF, Instituto Politécnico do Porto, Portugal
‡ RELEASE, Universidade da Beira Interior, Portugal
{jacome,jpaulo,prmartins,ruipereira,jas}@di.uminho.pt

***Abstract*—Software refactoring is a well-known technique that provides transformations on software artifacts with the aim of improving their overall quality. In this paper we present a set of refactorings for ClassSheets, a modeling language that allows to specify the business logic of a spreadsheet in an object-oriented fashion. The set of refactorings that we propose allows us to improve the quality of these spreadsheet models. Moreover, it is implemented in a setting that guarantees that all model refactorings are automatically carried to all the corresponding (spreadsheet) instances, thus providing an automatic evolution of the data so it is always synchronized with the model.**

I. Introduction

Software refactoring [1] is the process of modifying the structure of software programs without changing the way they behave. That is to say that while improvements are expected on the nonfunctional attributes of a piece of software, its is mandatory that its associated functional attributes must not be affected by refactorings.

The set of nonfunctional attributes of a software product include characteristics such as readability, maintainability and extensibility, and improvements are achieved, for example, by transforming it into a new version with: i) reduced complexity, ii) added expressiveness in either the code or its model (or both) or iii) diminished overall size (fewer methods, classes or lines of code), for example.

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, e.g. its increased readibility, traditional programming tasks become simpler and can be implemented faster.

Because of its applicability, code refactoring has been studied in different contexts, ranging from software source code [1], [2] or software models [3] to spreadsheets [4]. In this paper, we propose a series of refactorings for ClassSheets [5], a modeling language for spreadsheets.

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 methodology for dealing with spreadsheets, and namely to specify and maintain their business logic. This methodology envisions concrete spreadsheets (or spreadsheet instances) being automatically derived from, and maintained together with abstract specifications (or spreadsheet models) [5], [6]. This environment provides an efficient and effective model-driven spreadsheet development system. Indeed, errors can be prevented by carefully reasoning about, and designing, a concise model, instead of doing so with potentially large spreadsheets.

Being an essential artifact to a model-driven spreadsheet engineering environment, ClassSheet models may still suffer from the traditional problems of poor design and construction that have been found in other software artifacts. Thus, they may also benefit from the availability of a set of automated refactorings that improve their nonfunctional properties.

In this paper, we exploit this possibility: we propose a series of refactorings for the ClassSheets language. The catalog that we propose is inspired by the catalogue initially proposed by Martin Fowler in [1]: our refactorings consist of either i) refactorings that are straightforwardly derivable from Fowler's set or ii) refactorings that are inspired in Fowler's work, and in the work of some of his followers.

While changing the structure of a spreadsheet model, our refactorings do so without neither changing their behavior nor the business logic they implement. The work presented in this paper is intended to help in the process of constructing and managing spreadsheet models by the application of the refactorings.

Furthermore, the refactorings that we propose have been fully implemented under the spreadsheet development framework of [6], [7], [8]. This framework offers a bidirectional co-evolution setting where changes in an instance are reflected in the corresponding model and vice-versa. What this means for the refactorings that we present in this paper is that not only are we able to systematically refactor spreadsheet models, but we also perform the same disciplined and automatic transformations in all the instances it represents.

II. Model-Driven Spreadsheets

Engels *et al.* introduced the language ClassSheet [5] to leverage handling spreadsheets to a more conceptual level. Figure 1a shows a spreadsheet containing information about a small warehouse for a bar/coffee shop distribution. 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`, expanding vertically. 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 two both expand vertically.

In Figure 1b we can see an instance of the class from Figure 1a. 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 and Marco) and at last, four instances of `Product`.

On top of ClassSheets we have created MDSheet [6], a framework that provides a bidirectional ClassSheet ecosystem: the techniques and language described in that work allow transformations from model to be automatically applied to the instance and vice-versa, as illustrated in Figure 2.

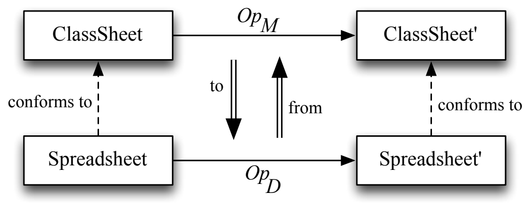


Fig. 2: Diagram of our bidirectional transformation system.

Given a spreadsheet that conforms 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 (data) is then automatically transformed into the corresponding operation on the data (model) using the *to* 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 reflect themselves on updates on the instance.

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

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

The first five operations are analogous to the data operations with the same name. Other operations include $setFormula_M$ which allows to define a formula on a particular cell. On the model side, a formula may be represented by an empty cell, by a default plain value (*e.g.*, an integer or a date) or by a function application. The operation $replicate_M$ allows to replicate (or duplicate) a class. The last two operations allow the addition of a new class to a model: $addClass_M$ adds a new static (non-expandable) class and $addClassExp_M$ creates a new expandable class. The $Direction$ parameter specifies if it expands horizontally or vertically.

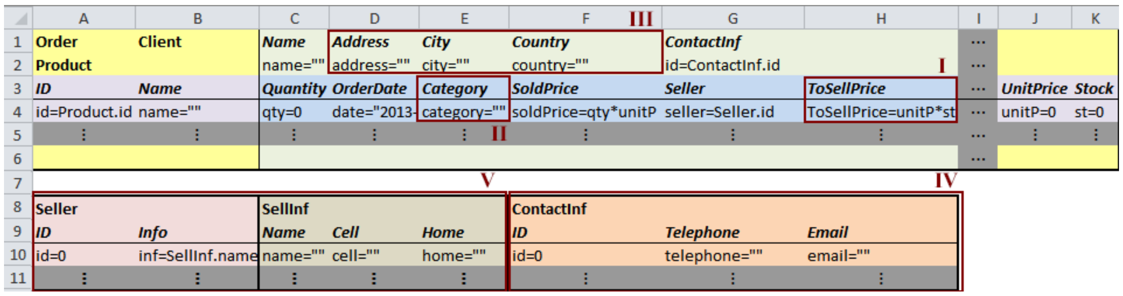
For more information on the MDSheet system please refer to [8]. In the next sections we will define a set of refactoring for ClassSheets that are based on this existing framework.

III. Model-Driven Spreadsheets Refactoring

We use a set of auxiliary functions to express our refactorings, 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. These functions are written using the ones defined before in *ModelOperation*.

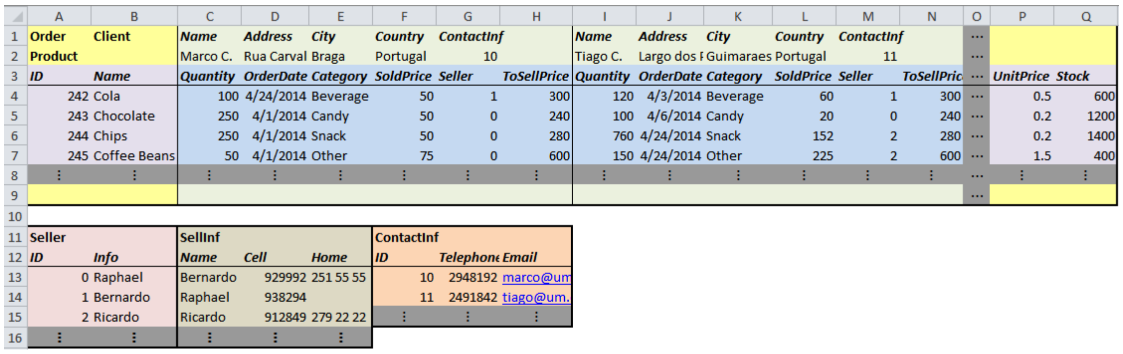
```
data ModelRefactoring : Spreadsheet → [ModelOperation] =
                                        -- add a formula
    AddShiftForm ClassName Value Index Label Index
                                        -- add an attribute
  | AddShiftAtt     ClassName Value Index Label Index
                                        -- delete a cell
  | DeleteShift     ClassName Value Label
                                        -- add a reference
  | AddShiftRef     ClassName ClassName Index
                                        -- delete a reference
  | DeleteShiftRef ClassName ClassName
                                        -- create a new class class
  | CreateClass     ClassName Direction Index
                                        -- delete a classs
  | DeleteClassShift ClassName
```

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.



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

(a) Classes.



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

(b) Instances.

Fig. 1: An example of classes and the respective instances for a warehouse goods distribution.

We will now present a set of refactorings for ClassSheets, how they apply to the models of Figure 1a, 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.

A. Move Formula

When/Why: We move formulas when they are more interested in and used by attributes of another class than the class on which they are defined. This is a phenomenon called *Feature Envy* [9].

If we closely analyze the `ToSellPrice` formula (shown in Figure 1a, 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 such 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. 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 Figure 3, 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 [9], [10], as the formula is now closer to the attributes it uses, and semantically in the correct class.

Evolution: The following steps describe the Move Formula refactoring:

```
Refactor - MoveFormula
Input: fromClass, value, label, toClass, posValue, posLabel
Output: [$ModelOperation$]
  DELETESHIFT(fromClass, value, label)
  ADDSHIFTFORM(toClass, value, posValue, label, posLabel)
```

To execute the Move Formula refactor on Figure 1a to obtain Figure 3 we would run:

```
MOVEFORMULA(Order, ToSellPrice, ToSellPrice, Product, L4, L3)
```

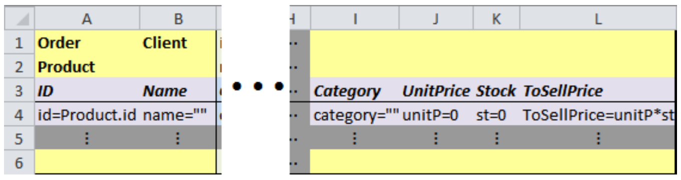
`MoveFormula` takes information from the class `Order`, namely the value `ToSellPrice` and the label `ToSellPrice` and moves them to the class `Product`, to the positions L3 and L4 respectively.

B. 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 attributes between classes due to information evolution.

Another reason would be in a relational class when we detect 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 [11], [12]. We can see a sample of this occur in Figure 1b - II on the `Category` attribute.

Refactoring: In the application of this refactoring 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 Figure 1a - II, we would move the `Category` attribute into the **Product** class, and obtain Figure 3 as our new class.



	A	B
1	Order	Client
2	Product	
3	*ID*	*Name*
4	id=Product.id	name=""
5	⋮	⋮
6		

I	J	K	L
Category	*UnitPrice*	*Stock*	*ToSellPrice*
category=""	unitP=0	st=0	ToSellPrice=unitP*st
⋮	⋮	⋮	⋮

Fig. 3: Move Formula and Move Attribute refactor on ToSellPrice and Category respectively

Evolution: The following steps describe the Move Attribute refactoring:

Refactor - MoveAttribute

```
Input: fromClass, value, label, toClass, posValue, posLabel
Output: [$ModelOperation$]
  DELETESHIFT(fromClass, value, label)
  ADDSHIFTATT(toClass, value, posValue, label, posLabel)
```

To execute the Move Attribute refactor on Figure 1a to obtain Figure 3 we would run:

```
  MOVEATTRIBUTE(Order, Category, Category, Product, I4, I3)
```

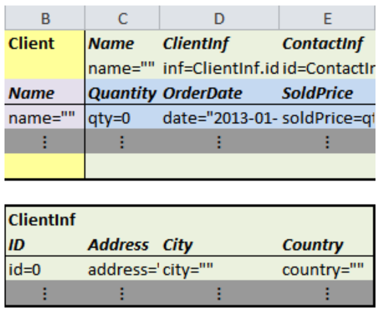
`MoveAttribute` moves the value `Category` and the label `Category` from the class `Order` to the class `Product`, more precisely to the positions `I4` and `I3`.

C. Extract Class

When/Why: Models can grow overtime due to 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 have a class doing the work of two.

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

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 would 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 Figure 4 from Figure 1a - III.



B	C	D	E
Client	*Name*	*ClientInf*	*ContactInf*
	name=""	inf=ClientInf.id	id=ContactIr
Name	*Quantity*	*OrderDate*	*SoldPrice*
name=""	qty=0	date="2013-01-	soldPrice=qt
⋮	⋮	⋮	⋮

ClientInf			
ID	*Address*	*City*	*Country*
id=0	address='	city=""	country=""
⋮	⋮	⋮	⋮

Fig. 4: Extract Class refactor on Address, City, and Country

Evolution: The following steps describe the Extract Class refactoring:

Refactor - Extract Class

```
Input: fromClass, newClass, newClassExp, newClassPos,
  list = [(value, label)]
Output: [$ModelOperation$]
  CREATECLASS(newClass, newClassExp, newClassPos)
  ADDSHIFTATT(newClass, 'id=0','ID')
  ADDSHIFTREF(fromClass, newClass)
  for all $(value, label) : list$ do
     DELETESHIFT(fromClass, value, label)
     if $value = formula$ then
        ADDSHIFTFORM(newClass, value, label)
     else
        ADDSHIFTATT(newClass, value, label)
     end if
  end for
```

To execute the Extract Class refactor on Figure 1a to obtain Figure 4 we would run:

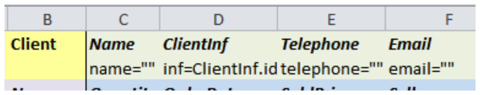
```
  EXTRACTCLASS(Client, ClientInf, Vertical, B8,
  [(Address,address),(City,city),(Country,country)])
```

`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 pairs (value,label) that will be extracted to the new class.

D. Inline Class

When/Why: Inline Class 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, we would remove the class, and join it with its outer-class or those which reference it.

***Refactoring**:* When we decide to use this refactor, we would choose the pointless class to apply this to. The attributes which existed in this pointless class would be transfered over to the referencing classes, replacing the referencing `ID` attribute, and eliminating the class in question. We can see this refactoring applied in Figure 1a - IV to obtain Figure 5.



B	C	D	E	F
Client	*Name*	*ClientInf*	*Telephone*	*Email*
	name=""	inf=ClientInf.id	telephone=""	email=""

Fig. 5: Inline Class refactor on the ContactInf class

***Evolution**:* The following steps describe the Inline Class refactoring:

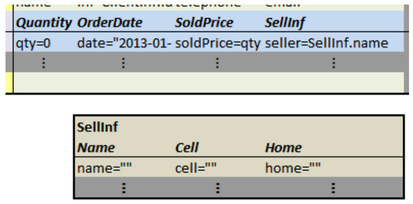
```
Refactor - Inline Class
Input: className
Output: [$ModelOperation$]
  for all $referencingClass : Spreadsheet$ do
    DELETESHIFTREF(referencingClass, className)
    for all $(value, label) : className$ do
      if value = formula then
        ADDSHIFTFORM(referencingClass, value, label)
      else
        ADDSHIFTATT(referencingClass, value, label)
      end if
    end for
  end for
  DELETECLASSSHIFT(className)
```

To execute the Inline Class refactor on Figure 1a to obtain Figure 5 we would run `Inlineclass`, which only has to receive as argument the name of the class `ContactInf`. This can be seen next:

```
INLINECLASS(ContactInf)
```

E. Remove Middle-Man

***When/Why**:* A middle-man smell is defined as a class which acts as a delegator between other classes. This delegator class does not usually contain enough responsibility, logic, or purpose other than the simple delegation of operations or information. Along with being insufficiently usefull, containing middle-mans usually complicate the structure and understanding of a spreadsheet.



Quantity	*OrderDate*	*SoldPrice*	*SellInf*
qty=0	date="2013-01-	soldPrice=qty	seller=SellInf.name
⋮	⋮	⋮	⋮

SellInf		
Name	*Cell*	*Home*
name=""	cell=""	home=""
⋮	⋮	⋮

Fig. 6: Remove Middle-Man refactor.

***Refactoring**:* When a middle-man exists it should be removed, and the classes connected to each other directly.

Looking at Figure 1a - V, we would remove the **Seller** class, which is doing absolutely nothing other than connecting to the **SellInf** class. We would then connect the **Order** class directly to the **SellInf** class, as shown in Figure 6.

***Evolution**:* The following steps describe the Remove Middle-Man refactoring:

```
Refactor - Remove Middle-Man
Input: className
Output: [$ModelOperation$]
  for all $referencingClass : Spreadsheet$ do
    DELETESHIFTREF(referencingClass, className)
    for all $referencedClass : className$ do
      ADDSHIFTREF(referencingClass,referencedClass)
      for all $(value, label) : className$ do
        if $value = formula$ then
          ADDSHIFTFORM(referencedClass, value, label)
        else
          ADDSHIFTATT(referencedClass, value, label)
        end if
      end for
    end for
  end for
  DELETESHIFTCLASS(className)
```

To execute the Remove Middle-Man refactor on Figure 1a to obtain Figure 6 we would run:

```
REMOVEMIDDLEMAN(Seller)
```

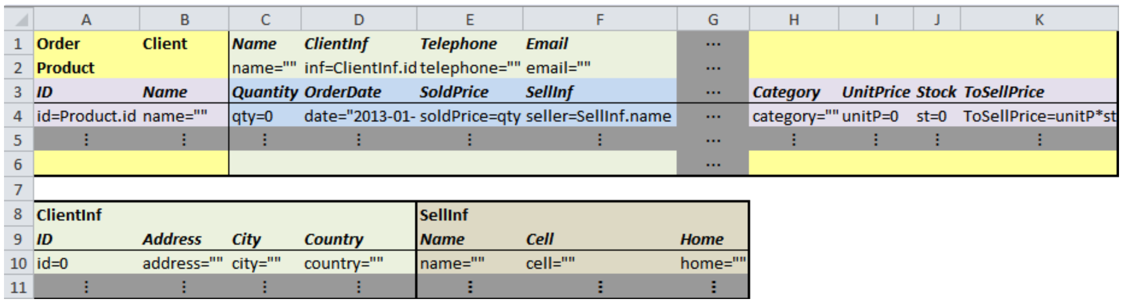
F. Refactored Example

Figure 7a shows a refactored ClassSheet model. We were able to remove one useless class, and organize the data to be semantically correct. The refactorings also made it easier for the user to not only read, but use the spreadsheet more efficiently by joining attributes closer to their formulas, and placing often used attributes in classes easier to access (such as the joining of the Client's email and telephone into the **Client** class). Figure 7b shows the co-evolved spreadsheet instance, still in conformity with our new model.

IV. Related Work

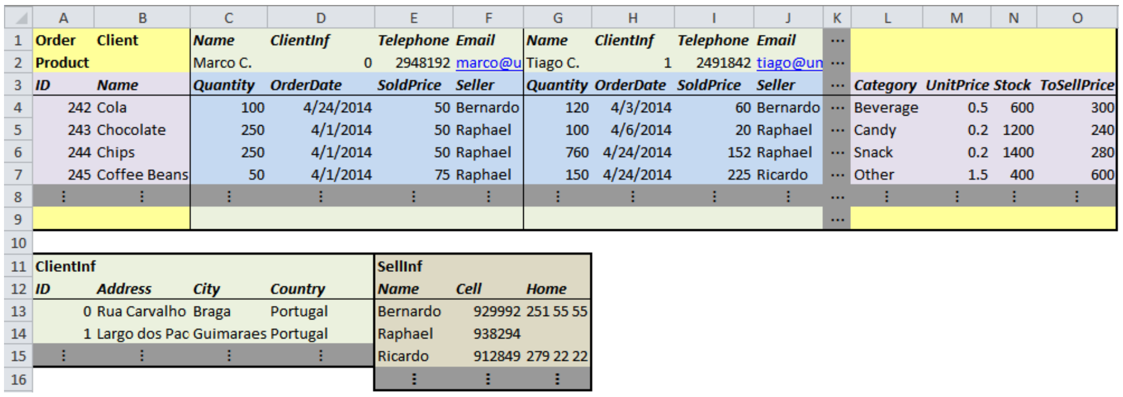
Previous works have focused on refactoring spreadsheets formulas. Of special mention is [4], where the authors suggest a set of transformations to formulas using an Excel plugin. The fundamental difference to our work is that the refactorings we propose are applied to the spreadsheet model, which being more concise, makes the reasoning easier. WYSIWYG [13], [14] 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.* [9], [15] have various works on spreadsheet smells. They sometimes refer refactorings for the smells they introduce, but they focus on detection, not correction. Their work is complementary to ours, as they focus on detecting spreadsheet problems and ours on solving them.



	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=Product.id	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	⋮	⋮	⋮	⋮	⋮	⋮	⋮				

(a) Classes after refactoring.



	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					⋮	⋮	⋮								

(b) Instances after refactoring.

Fig. 7: The classes and the respective instances after refactoring the ClassSheet and having the instance automatically co-refactor

V. Conclusions and Future Work

This paper presents a set of refactorings for ClassSheets. These provide better models, which are easier to understand and to reason about. The refactorings have been implemented in a tool, ensuring the automated application of model refactorings, and their propagation to the corresponding instances such that model/instance synchronization is guaranteed. As for future work, we will look into expanding our catalog of ClassSheet refactorings and validating the existing one, with the help of quality assessment metrics [16] and through empirical studies with end users.

References

[1] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Aug. 1999.

[2] T. Mens and T. Tourwe, "A survey of software refactoring," *IEEE Transactions on Software Engineering*, vol. 30, no. 2, 2004.

[3] H. T. Einarsson and H. Neukirchen, "An approach and tool for synchronous refactoring of UML diagrams and models using model-to-model transformations," in *Proc. of the Fifth Workshop on Refactoring Tools*, ser. WRT '12. ACM, 2012.

[4] S. Badame and D. Dig, "Refactoring meets spreadsheet formulas," in *Proc. of the 2012 IEEE International Conference on Software Maintenance (ICSM)*, ser. ICSM '12. IEEE Computer Society, 2012.

[5] G. Engels and M. Erwig, "ClassSheets: automatic generation of spreadsheet applications from object-oriented specifications," in *Proc. of the 20th IEEE/ACM international Conference on Automated software engineering*. ACM, 2005.

[6] J. Cunha, J. P. Fernandes, J. Mendes, and J. Saraiva, "MDSheet: A framework for model-driven spreadsheet engineering," in *Proc. of the 2012 International Conference on Software Engineering*. IEEE Press, 2012.

[7] ——, "A bidirectional model-driven spreadsheet environment," in *Proc. of the 2012 International Conference on Software Engineering*. IEEE Press, 2012.

[8] J. Cunha, J. P. Fernandes, J. Mendes, H. Pacheco, and J. Saraiva, "Bidirectional transformation of model-driven spreadsheets," in *Theory and Practice of Model Transformations*. Springer, 2012.

[9] F. Hermans, M. Pinzger, and A. v. Deursen, "Detecting and visualizing inter-worksheet smells in spreadsheets," in *Proc. of the 2012 International Conference on Software Engineering*. IEEE Press, 2012.

[10] D. Conway and C. Ragsdale, "Modeling optimization problems in the unstructured world of spreadsheets," *Omega*, vol. 25, no. 3, 1997.

[11] D. Maier, *The Theory of Relational Databases*. Computer Science Press, 1983.

[12] J. Cunha, J. Saraiva, and J. Visser, "From spreadsheets to relational databases and back," in *Proc. of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program manipulation*, 2009.

[13] M. Fisher II, M. Cao, G. Rothermel, D. Brown, C. Cook, and M. Burnett, "Integrating automated test case generation into the WYSIWYT spreadsheet testing methodology," Oregon State University, Corvallis, OR, USA, Tech. Rep., Feb. 2002.

[14] K. J. Rothermel, C. R. Cook, M. M. Burnett, J. Schonfeld, T. R. G. Green, and G. Rothermel, "WYSIWYT testing in the spreadsheet paradigm: An empirical evaluation," in *Proc. of the 22Nd International Conference on Software Engineering*, ser. ICSE '00. ACM, 2000.

[15] M. Pinzger, F. Hermans, and A. van Deursen, "Detecting code smells in spreadsheet formulas," in *Proc. of the 2012 IEEE International Conference on Software Maintenance (ICSM)*, ser. ICSM '12. IEEE Computer Society, 2012.

[16] J. Cunha, J. P. Fernandes, J. Mendes, and J. Saraiva, "Complexity metrics for classsheet models," in *Computational Science and Its Applications–ICCSA 2013*. Springer, 2013.