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 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 readability, 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 $op_M m : Model$.

The first five operations are analogous to the data operations with the same name. Other operations include setFormula, 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 allows to replicate (or duplicate) a class. The last two operations allow the addition of a new class to a model: addClass adds a new static (non-expandable) class and addClassExp adds 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 REFACCTORING

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.

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

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 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:* 

```plaintext
DELETE SHIFT(fromClass, value, label)
ADD SHIFT FORM(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 instance 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.

![Fig. 3: Move Formula and Move Attribute refator on ToSell-Price and Category respectively](image)

**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 refator 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 subset of client information, and make reading the Client class difficult, it is a good candidate for Extract Class.

**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, value, label, posLabel)
- 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 transferred 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.

Fig. 5: Inline Class refactor on the ContactInf class

Evolution: The following steps describe the Inline Class refactoring:

Refactor - Inline Class
Input: className
Output: 
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
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.

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
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.
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