JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2007**

Vol. 6, No. 9, Special Issue: TOOLS EUROPE 2007, October 2007

ClassSheets - model-based, object-oriented design of spreadsheet applications

Jan-Christopher Bals, Software Quality Lab, University of Paderborn, Germany
Fabian Christ, Software Quality Lab, University of Paderborn, Germany
Gregor Engels, Workgroup Database and Information Systems, University of Paderborn, Germany
Martin Erwig, School of Electrical Engineering and Computer Science, Oregon State University, USA

Abstract

Using spreadsheets is the preferred method to calculate, display or store anything that fits into a table-like structure. They are often used by end users to create applications. But they have one critical drawback - they are very error-prone.

To reduce the error-proneness, we purpose a new way of object-oriented modeling of spreadsheets prior to using them. These spreadsheet models, termed *ClassSheets*, are used to generate concrete spreadsheets on the instance level.

By this approach sources of errors are reduced and spreadsheet applications are easier to understand.

1 INTRODUCTION

Spreadsheets are widely used in all kinds of businesses applications [EE05]. Using spreadsheet is the preferred method to calculate, display or store anything that fits into a table-like structure. The application area of spreadsheets seems to be unlimited. Spreadsheets are flexible to use, easy to create, and available on almost every PC on which an office suite is installed.

Because of their flexibility spreadsheets are a standard tool in offices. Knowledge of office software, including spreadsheets, is a requirement when someone applies for a job in today's offices. Every year business people create hundreds or thousands of spreadsheets [Panko00]. Thus, spreadsheets are the most popular programming system

Cite this article as follows: J.-C. Bals, F. Christ, G. Engels and M. Erwig: “ClassSheets - model based, object oriented design of spreadsheet applications”, in *Journal of Object Technology*, Special Issue: TOOLS EUROPE 2007,October 2007 http://www.jot.fm/issues/issue_2007_10/paper19

used today. But both the application of spreadsheets and the knowledge of spreadsheet users about programming vary. A great amount of users don't have or just have little knowledge about software development or programming, but they use spreadsheets to program applications. The creation of spreadsheet applications is end-user programming. In contrast to a professional, end users don't have the knowledge of preventing errors when programming a spreadsheet application. Many errors in applications can be referred to this fact [Boehm01].

First it seems that spreadsheets are easy to use and the basics can be learned fast. The reason for that is the simplicity of spreadsheets with their two-dimensional tabular layout. But even small spreadsheet examples show how error-prone spreadsheet applications are, and this observation is enforced when big spreadsheets with hundreds or thousands of formulas and datasets are examined [Eusprig, PB04].

Users are often forced to correct formulas and the calculation itself just for inserting a new row with a new dataset. When a user copies a formula from one cell to another without adjusting it afterwards, such mistakes might result in wrong calculations. Error messages or warnings are often misleading as the spreadsheet software can only guess what the user wants. The lack of information about what the user tries to model makes it impossible for spreadsheet software to support the user in an effective way, e.g. to show mistakes or to prevent error situations.

How dramatic the consequences of a simple cut-and-paste error in a spreadsheet application can be shows the following real life example:

"June 03, 2003 TORONTO (Reuters) - TransAlta Corp. said on Tuesday it will take a $24 million charge to earnings after a bidding snafu landed it more U.S. power transmission hedging contracts than it bargained for, at higher prices than it wanted to pay.

[...] the company's computer spreadsheet contained mismatched bids for the contracts, it said. "`It was literally a cut-and-paste error in an Excel spreadsheet that we did not detect when we did our final sorting and ranking bids prior to submission,'" TransAlta chief executive Steve Snyder said in a conference call. "`I am clearly disappointed over this event. The important thing is to learn from it, which we've done.'"

As New York ISO rules did not allow for a reversal of the bids, the contracts went ahead." [Eusprig]

Thus, the problem to be solved is to minimize the gap between what the user tries to model and what the spreadsheet software knows.

The object-oriented (oo) paradigm improved the design and development of software. Programming languages evolved from assembler over languages with macros and later functions to the point of object oriented languages. To fulfill today's needs, the design of spreadsheet applications has to evolve similarly.

We propose a new way of oo-modeling of spreadsheets prior to using them. These spreadsheet models, termed *ClassSheets*, are used to generate concrete spreadsheets on the instance level. The following figure illustrates this idea.

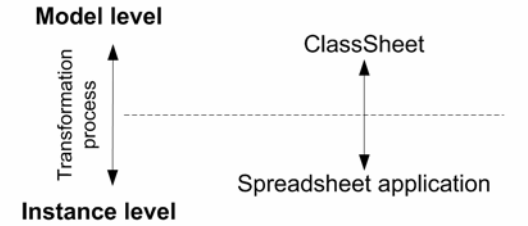

Figure 1: Model and instance level of ClassSheets

The idea and formalism of ClassSheets was published by Engels and Erwig in [EE05]. ClassSheets give an oo-view at the modeling level on spreadsheets. The modeling of ClassSheets becomes an integrated step in the development of spreadsheet applications. The spreadsheet software uses the information from the ClassSheet to support the user, e.g. to insert new datasets, to prevent error situations, and to ensure the consistency between model and spreadsheet. ClassSheets provide oo-model-driven spreadsheet design to create spreadsheets with less errors and better user support.

The oo-modeling paradigm increases the readability of spreadsheets and makes them easier to understand than traditional ones. ClassSheets use classes with unique names to group information. Associations between information groups are expressed as associations between classes. The use of unique names for classes and attributes makes it possible to reference attributes simply by writing `ClassName.AttributeName`. Cell references in functions known by classical spreadsheets like `SUM(A1:D6)` are not used any more. The distinction between relative and absolute cell references like `A1` to `$A$1` is not needed as attribute references are always unique.

In this paper, we focus on two aspects related to the work of Engels and Erwig [EE05]. At first, we introduce how to model correct spreadsheet applications with ClassSheets and provide tool support for this task. *Claos* is a prototypical ClassSheet editor developed by the authors and is used to demonstrate the work with ClassSheets. Secondly, we briefly describe an enhanced version of the ClassSheet formalism [BC06], which consists of a ClassSheet syntax and a set of rules, called tiling inference rules, to verify well-formed ClassSheets.

The example

To demonstrate the modeling technique using ClassSheet we take a look at an example, which gives us the following situation:

» A company's sales department is interested in the profit they realize for each produced product. Those products are sold in different countries. The profit is calculated for one sold product. Products are produced at given costs and achieve different prices per country. The company needs a spreadsheet application that shows all products and their production costs. For each country, the application has to store the price of sale and calculates the profit for each product in that country. The last step is to calculate the total costs for all products, the profit per country, the profit per product and the total profit. «

OO-design of the profit example

We model the given example by using an UML class diagram. The UML model is only used to illustrate the oo-characteristics of the example and to show how the oo-design is supported by ClassSheets. ClassSheets itself don't use any UML diagrams. When analyzing the given situation, we identify four classes: Sales, Product, Country and Product per Country (PxC)

Furthermore, we identify some properties: Sales knows the total cost and the total profit, each Product has a different price in each Country and the profit is calculated for each Country and Product. The UML class diagram is shown in Figure 2.

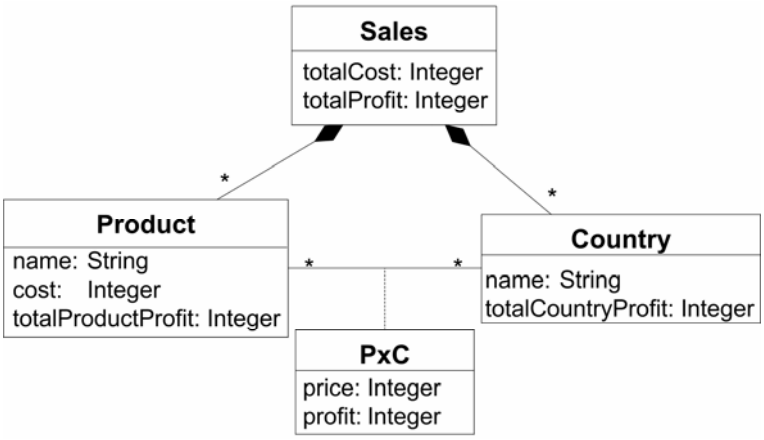

Figure 2: UML class diagram of the profit example

In this example Sales consists of $0 \ldots n$ Products and Countries. The association class between Product and Country stores the price and profit for each Product per Country. The following OCL constraints [OMG03] define the different properties.

```
context: Product
def: totalProductProfit: Integer =
                          self.pxC[Country].profit->sum()
context: Country
def: totalCountryProfit: Integer =
                          self.pxC[Product].profit->sum()
context: Sales
def: totalCost: Integer = self.product.cost->sum()
def: totalProfit: Integer =
                   self.product.totalProductProfit->sum()
```

The profit ClassSheet

When modeling the example as a ClassSheet we have two things in mind

- the oo-model and class structure, respectively, and
- the cell layout of the subsequent spreadsheet.

Given the UML class diagram and some OCL constraints, we have to integrate the oo-design with cell layout information by using ClassSheets. In relation to the example the class Sales is represented by Figure 3.

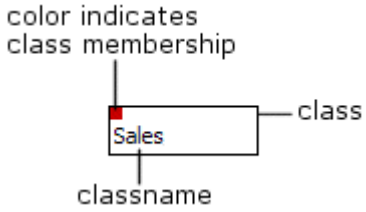

Figure 3: Sales class

Figure 4: Sales class encapsulates Product class

The layout representation of a class is a block of cells. When modeling the class structure of a ClassSheet the single cells aren't displayed as they are of no interest in this view. Cells of classes are only displayed in the cell layout view.

Classes can be arranged in horizontal or vertical manner to express associations between classes. They can be arranged side by side either horizontally or vertically, or one can encapsulate another to express a 'consists of' relationship. Since Sales consists of $0\ldots n$ Products, Sales encapsulates Products, e.g. vertically as Figure 4 shows.

Since Sales also consists of $0\ldots n$ Countries that also have to be encapsulated by Sales. As Product is already vertically inserted, the Country class is horizontally inserted. During the insert, a fourth class arises as the intersection of Product and Country. This class is Product × Country (*PxC*) and is the association class between Product and Country. The result is a table-like class structure.

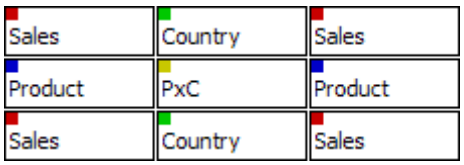

Figure 5: Table-like class structure

Having modeled the class structure and their relationships, we still have to define attributes corresponding to the UML model and the cell layout. The definition of an attribute comprises the following four aspects.

1. The attribute's class
2. The attribute's name
3. The attribute's cell in which it will be displayed
4. The attribute's kind: constant, reference or function

The definition of attributes and the definition of the cell layout is an integrated step. The cell layout depends on the class structure. A class consists of a rectangular block of cells. When combining classes horizontally or vertically, the blocks of cells are combined. Similarly, the block of cells of one class encapsulates the block of cells of another class if the class encapsulates the other. Thus, each cell belongs to exactly one class. Three kinds of cells exist:

- empty cells
- label cells
- attribute cells

Empty cells contain nothing and are only used for layout purposes. Label cells are used as captions. An attribute cell contains a class's attribute name and its value. There are three kinds of values:

- constant values
- reference values
- values calculated by a function

A constant attribute has an initial value and can be modified by the user at the instance level. Attributes with references or functions are read-only in the later application as the ClassSheet model defines how those values are calculated. To alter references or functions the user has to edit the ClassSheet at the modeling level.

Sales has the attribute totalCost, which is calculated as the sum over all Product costs. In ClassSheets this is expressed by:

```
Sales.TotalCost = SUM(Product.Cost)
```

The attributes name and cost of Product are constant values which get the initial values of `name` and `0`.

```
Product.Name = name
Product.Cost = 0
```

As seen, a reference can be expressed by using the class name and the attribute name concatenated by a dot. An attribute reference definition looks like this:

```
ClassName.AttributeName = AnotherClassName.AttributeName
```

Back to the example, we take a look at the classes Sales and Product. Sales consists of $0 \ldots n$ Products. The cardinality of associations is expressed by defining recurrences of classes and their corresponding cells. The cardinality $0 \ldots n$ is set by marking the Product class as vertically repeatable. The recurrence has to be specified at class level and in the cell layout. The cell layout defines which rows or columns of a class are repeated. Figure 6 shows the class structure and the cell layout of the class Sales with inner class Product.

Only row number three of class Product is marked as vertically repeatable (indicated by the small down-arrows at the bottom right corner of the cells). Row number two is fixed and used as a caption row in the latter spreadsheet application.

Like the Product class is vertically repeatable, the Country class is horizontally repeatable as Sales consists of $0 \ldots n$ Countries. The association class PxC inherits both recurrences and is vertically and horizontally repeatable. The resulting spreadsheet application can grow vertically by inserting new products as rows and horizontally by adding new countries as columns.

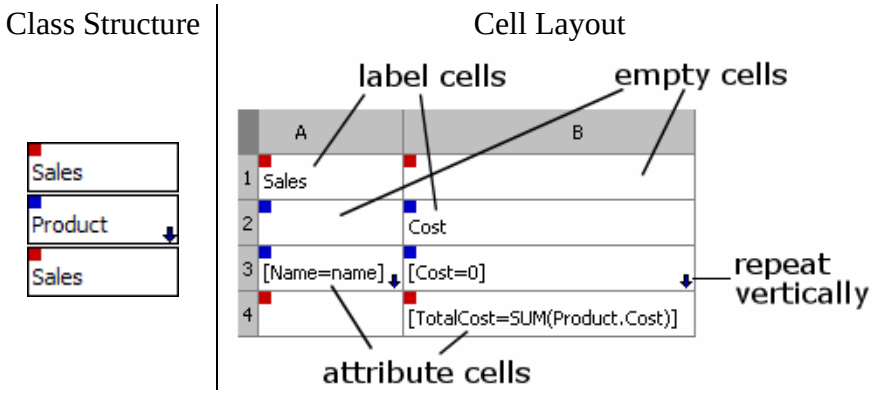

Figure 6: Sales/Product class structure and cell layout

The spreadsheet software handles the insertion of rows and columns by using the information taken from the ClassSheet model. All functions and references are correctly updated when new datasets are inserted. Classical copy & paste user operations aren't needed anymore and possible error situations are prevented.

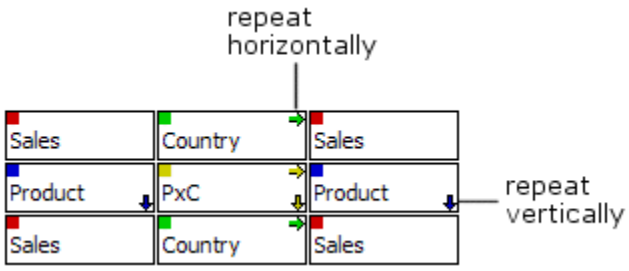

Figure 7: Table-like class structure of the example

Above, Figure 7 shows the table-like class structure of the example with recurrences and below, Figure 8 the cell layout of the profit example.

	A	B	C	D	E
1	Sales		Country	[Name=name]	
2		Cost	Price	Profit	
3	[Name=name]	[Cost=0]	[Price=0]	[Profit=Minus(PxC.Price,Product.Cost)]	[TotalProductProfit=SUM(PxC.Profit)]
4		[TotalCost=SUM(Product.Cost)]		[TotalCountryProfit=SUM(PxC.Profit)]	[TotalProfit=SUM(Product.TotalProductPr

Figure 8: ClassSheet of the profit example

2 THE TOOL CHAIN

The modeling of ClassSheets is supported by a two-step tool chain. The first step is to design and model ClassSheets like shown in the example. We implemented a prototypical ClassSheet editor called *Claos* [BC06]. It is a standalone editor to create and modify ClassSheet models. Claos is no spreadsheet software like Microsoft Excel. To use the spreadsheet applications that are generated from ClassSheets we need a tool chain to make ClassSheets usable in spreadsheet software, e.g. Microsoft Excel.

ClassSheet specifications are translated into spreadsheet templates in a language called *Gencel* [EE05, EACK05b] that capture the evolution of spreadsheet instances. Gencel is an extension for Microsoft Excel that enables block manipulation actions on spreadsheets. A block is analogue to a class in ClassSheets a rectangular composition of cells. Gencel ensures that the spreadsheet stays within the model defined by its input [EACK05a]. A visual representation, called ViTSL (Visual Template Specification Language) [AECK04], was defined to support the interactive editing of templates. In this sense ViTSL served a similar purpose to ClassSheets. However, an advantage of ClassSheets is that they contain higher-level information, such as class- and attribute names.

The second step in the tool chain is to generate the spreadsheet application from the ClassSheet model as a Gencel template. Claos offers an export of the block structure of a ClassSheet in Gencel/ViTSL format.

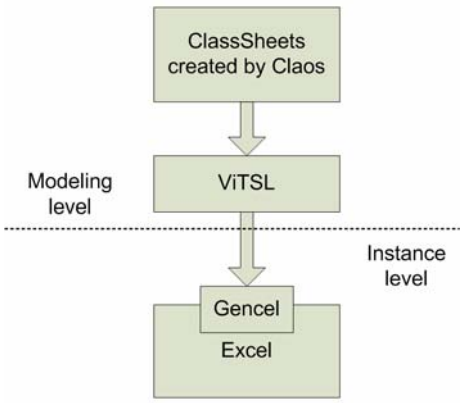

Figure 9: The tool chain

Figure 1 illustrates the complete tool chain. The ClassSheet model is exported in ViTSL format which is read by Gencel that enables Excel to handle block structures of the resulting spreadsheet application. The possibility to handle block structures makes it possible to use the powerful construct of repeating blocks in Excel.

Claos - The ClassSheet editor

The ClassSheet editor Claos is a prototypical development to demonstrate the usability of our approach. Claos implements four use cases:

- Handle ClassSheet packages
- Edit class structure
- Edit cell layout
- Export in ViTSL format

Claos is written in Java and is based on the Eclipse Rich Client Platform (RPC). The GUI is grouped into three views:

- Package Explorer
- Class Outline
- Layout Editor

The package explorer offers operations to store and load packages of ClassSheets. A ClassSheet package contains several ClassSheets and is the format to store ClassSheets. All classes and their structure are displayed in the class outline view. The GUI layout is the same as seen in the profit example, e.g. see Figure 5.

The third view is the layout editor. The cell layout of the latter spreadsheet application is defined by the editor. A user can add new rows or columns, delete them and edit label cells. There is one editor instance for each opened ClassSheet. All opened ClassSheets can be accessed by the editor's tabs at the top of the editor view. Again the GUI layout is the same as seen during the profit example, e.g. see Figure 8.

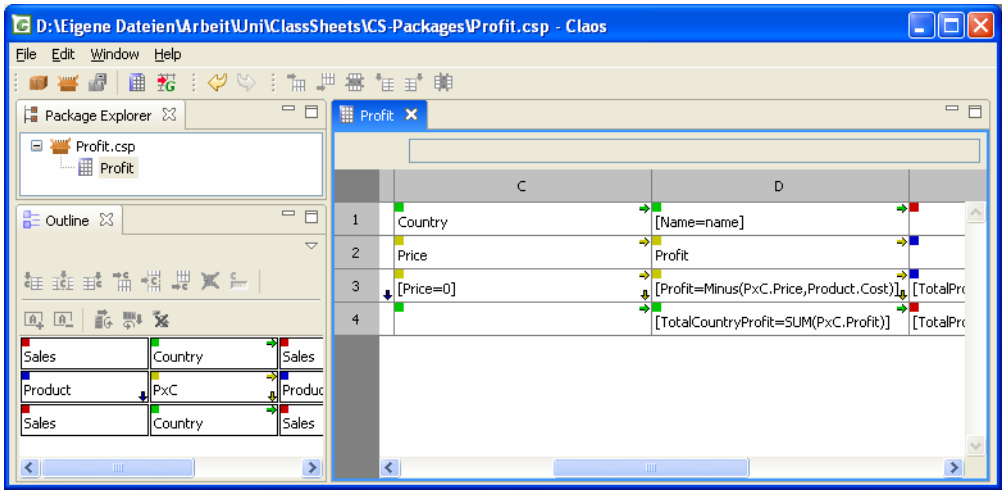

Figure 10: Profit example in Claos

The workflow in Claos is similar to the steps made for the profit example. At first the class structure, then the cell layout and the class's attributes are defined. At last vertical or horizontal recurrences are specified. To use the spreadsheet application the ClassSheet is exported in ViTSL format to make it loadable by Gencel.

Gencel

Gencel [EACK05b] is an extension for Microsoft Excel that offers block manipulation functions. A block structure is a rectangular composition of cells that may be repeatable. Blocks can be grouped together horizontally or vertically. A block is similar to a class in ClassSheets without name and attributes. The tool loads spreadsheet applications in ViTSL format. Gencel extends Microsoft Excel by these new functions:

- Open a ViTSL file.
- Insert new blocks that were defined as horizontally or vertically repeatable.
- Delete blocks that were inserted before.
- Update references and functions that have to change after an insert or delete operation.

The following series of screenshots show how to work with Gencel and the generated profit spreadsheet application. Figure 11 shows an example with one product and one country. To add a second product we have to insert a row below the existing product row.

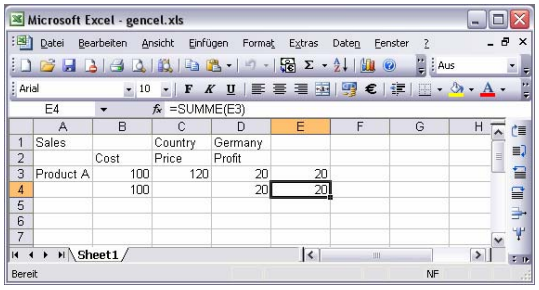

Figure 11: Profit spreadsheet application

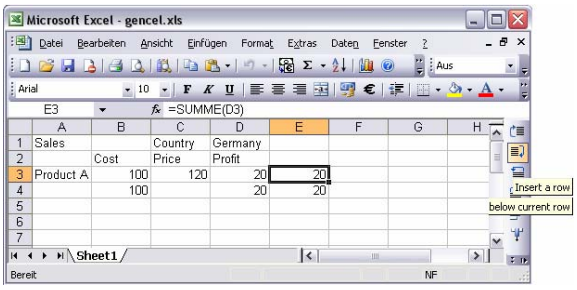

Figure 12: Vertically insert new Product

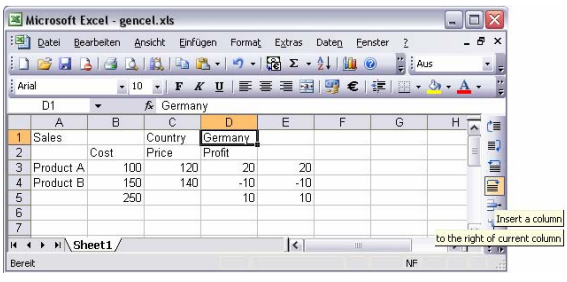

Figure 13: Horizontally insert new Country

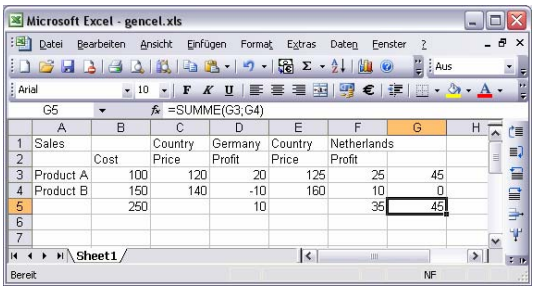

Figure 14: Two Products and Countries

As Gencel only processes block structures it has no information about the class Product. But, Gencel knows how to insert a new row for the block structure defined by Product. The user just needs to update the values in the different cells: the product name, the costs and the prizes for each country. Figure 13 shows the example with a newly inserted product. Now we add a second country by adding a new block structure to the right of the existing country entry. All formulas and references are correctly updated. The user just

enters the country name and the prizes for each product. The sums of total costs and profit are automatically and correctly calculated as Figure 14 shows.

3 A FORMAL MODEL OF CLASSSHEETS

The formal model of ClassSheets was introduced in [EE05] and enhanced in [BC06].

The syntax for ClassSheets, shown in Figure 15, defines a class as a composition of blocks. One block in the syntax is one cell of a ClassSheet. Blocks can be arranged horizontally or vertically and compose the class's layout.

f	$\in$	Fml	$::=$	$\varphi \mid n.a \mid \varphi(f,\ldots,f)$	(formulas)
b	$\in$	$Block$	$::=$	$\varphi \mid a = f$	(blocks)
α	$\in$	$horExp$	$::=$	$b \mid b \vert \alpha$	(horizontal expansion)
β	$\in$	$verExp$	$::=$	$\alpha \mid \alpha \hat{} \beta$	(vertical expansion)
l	$\in$	Lab	$::=$	$h \mid v \mid .n$	(class labels)
h	$\in$	Hor	$::=$	$\underline{n} \mid \vert \underline{n}$	(horizontal)
v	$\in$	Ver	$::=$	$\vert n \mid \overline{\vert n}$	(vertical)
c	$\in$	$Class$	$::=$	$l : \beta \mid l : \beta^{\downarrow} \mid c \hat{} c$	(classes)
s	$\in$	$Sheet$	$::=$	$c \mid c^{\rightarrow} \mid s \vert s$	(sheets)

Figure 15: ClassSheet syntax

$\updownarrow \alpha$	$=$	1	$\overleftrightarrow{b}$	$=$	1
$\updownarrow \beta$	$=$	$\updownarrow \alpha_1 + \updownarrow \beta \backslash \{\alpha_1\}$	$\overleftrightarrow{\alpha}$	$=$	$\overleftrightarrow{b_1} + \overleftrightarrow{\alpha \backslash \{b_1\}}$

Figure 16: Enhanced calculation of width and height of expansions

The syntax allows the construction of classes that may be rectangular, but they do not have to be. Two horizontally composed classes must have the same height. Accordingly two vertically composed classes must have the same width. The ClassSheet syntax doesn't ensure that those requirements are fulfilled, that is, the set of ClassSheets that can be expressed by the syntax is bigger than the set of *well-formed* ClassSheets. Therefore, we use a second grammar and rules, termed *tilings* and *tiling rules*, to check whether a given ClassSheet is well-formed.

Before we introduce tilings in detail we show an enhanced version of formulas to calculate the width and height of expansions measured in blocks. We use the ones shown in Figure 16.

The tiling of a ClassSheet gives us information about the construction of classes and how classes are composed. Additionally, define the tiling rules how to retrieve a tiling for a given ClassSheet. A tiling can be expressed by the syntax in Figure 17.

The tiling syntax defines four kinds of tilings for a well-formed ClassSheets:

- *horizontal-tiling* for class structures in which classes are only horizontally composed side by side
- *vertically-tiling* for class structures in which classes are only vertically composed among each other
- *table-tiling* for class structures in which classes are both horizontally and vertically composed.

In Figure 18 we show an enhanced formalism to express the height and width of class compositions in the tiling rules.

The formalism in [EE05] was not strict enough and allowed the construction of ClassSheets with incorrect tilings. The tiling rules definee the inference of tilings for a given ClassSheet.

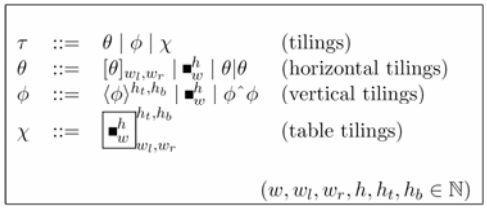

Figure 17: Enhanced tiling syntax

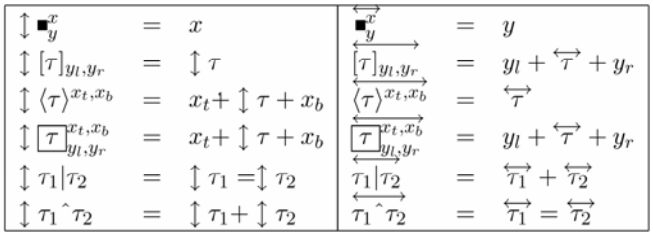

Figure 18: Enhanced calculation of tiling height and width

The enhanced tiling rules ensure that a tiling is only inferred if the given ClassSheet is well-formed. To calculate whether a tiling has correct height and width corresponding to the composition of blocks and classes the formulas shown in Figure 19 are used.

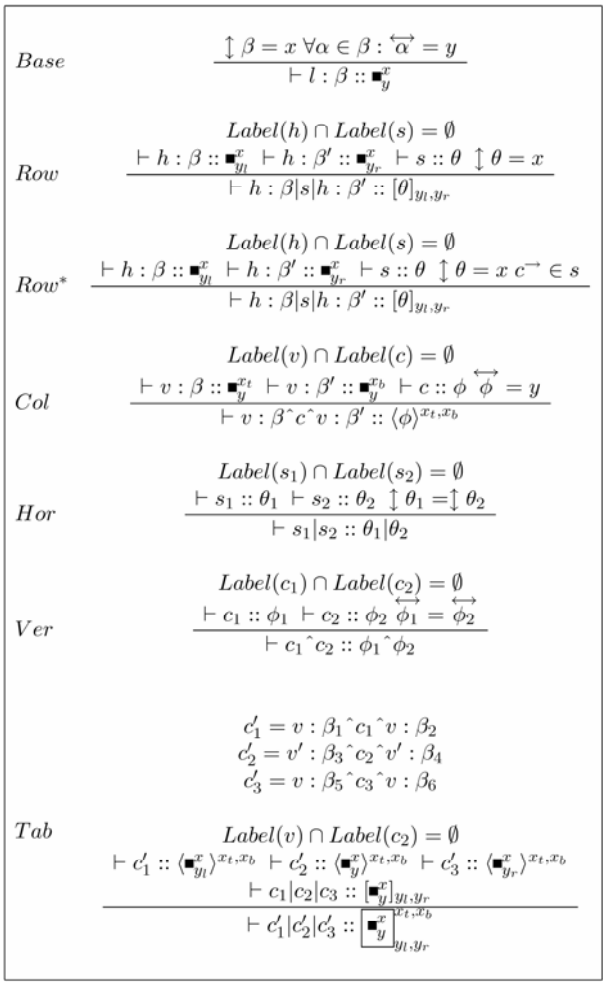

Figure 19: Enhanced tiling rules

A ClassSheet is well-formed if the application of the tiling rules results in a tiling that is valid according to the tiling syntax. Claos implements the tiling rules to ensure that only well-formed ClassSheets can be created. Each user operation that performs a ClassSheet modification requires a tiling inference. Only if this inference succeeds and results in a valid tiling, the user operation is accepted. Otherwise, the operation is rejected or it is inaccessible to the user. The tiling enhancements have thus a direct impact on the implementation of the ClassSheet editor. Back to our example, we show the profit ClassSheet in ClassSheet syntax below.

```
|Sales:    Sales       | blank                                ^
|Product:  _           | Cost                                 ^
|Product:  Name=name↓  | Cost=0↓                              ^
|Sales:    _           | TotalCost=Sum(Product.Cost)
|
(Country:  Country     | Name=name                            ^
```

```
  .PxC:    Price      | Profit                                    ^
  .PxC:    Price=0↓   | Profit=Minus(PxC.Price,                   
                                    Product.Cost)↓                ^
Country:   _          | TotalCountryProfit=Sum(PxC.Profit))→
|
 |Sales:   _                                                      ^
 |Product: _                                                      ^
 |Product: TotalProductProfit=Sum(PxC.Profit)↓                    ^
 |Sales:   TotalProfit=Sum(Product.TotalProductProfit)
```

ClassSheet of the profit example in ClassSheet syntax

To verify that the ClassSheet is well-formed we apply the tiling rules. The resulting tiling of the example is a table-tiling. Thus, the modeled ClassSheet is well-formed.

4 CONCLUSION

ClassSheets transfer the oo-design paradigm known from programming languages to the world of spreadsheet applications. The new modeling level improves the design and correctness of spreadsheet applications. The formalization of well-formed ClassSheets through a rule system guarantees the generation of spreadsheet applications that evolve in a predictable way.

Spreadsheet software can offer better user support using the information from ClassSheet models. Error situations are prevented, which improves the quality of spreadsheet applications. Error-prone user operations like the copy & paste of formulas are not needed anymore. References and formulas are automatically updated by the spreadsheet software when new datasets are inserted.

With ClassSheets the oo way of thinking is established at the level of end-user programming. ClassSheets are designed to be used by end users. Complicated UML class diagrams aren't needed by end users. Instead, the ClassSheets' oo-design helps them to model and program their applications. Spreadsheet applications with ClassSheet information carry more model information and are therefore easier to understand by users who know about the modeling level. The design can also become more comprehensible for others.

We have introduced a tool chain that makes it possible for the first time to use ClassSheets in conjunction with Microsoft Excel. The tool chain uses Gencel to bridge the gap between Claos and Excel. The ClassSheet modeling approach provides an integrated method for the creation of spreadsheet applications. However, the integration of ClassSheets with Excel is not complete at this moment. For example, the class structure cannot be used or displayed within Excel. We are currently investigating mechanisms for a close coupling between ClassSheets and Excel that allows, in

particular, the automatic propagation of changes from ClassSheet models to spreadsheet instances.

REFERENCES

[AEKS04] R. Abraham, M. Erwig, S. Kollmansberger and E. Seifert: "Visual Specifications of Correct Spreadsheets.", *IEEE Int. Symp. on Visual Languages and Human-Centric Computing*, pp. 165-172, 2004.

[BC06] J.-C. Bals and F. Christ: "ClassSheets – modellbasierter, werkzeuggestützter Entwurf von Spreadsheet-Anwendungen", in German, July 2006.

[BB01] B. Boehm and V. R. Basili: "Software Defect Reduction Top 10 List". *Software Management*, pp. 135–137, January 2001.

[EE05] G. Engels and M. Erwig. "ClassSheets: Automatic Generation of Spreadsheet Applications from Object Oriented Specifications", *20th IEEE/ACM International Conference on Automated Software Engineering*, pp. 124-133, 2005.

[EACK05a] M. Erwig, R. Abraham, I. Cooperstein and S. Kollmansberger: "Automatic Generation and Maintenance of Correct Spreadsheets", *27th IEEE Int. Conf. on Software Engineering*, pp. 136-145, 2005.

[EACK05b] M. Erwig, R. Abraham, I. Cooperstein and S. Kollmansberger: "Gencel - A Program Generator for Correct Spreadsheets", *Journal of Functional Programming*, vol. 16, no. 3, pp. 293-325, May 2006.

[Eusprig] European Spreadsheet Risks Interest Group (EuSpRIG). http://www.eusprig.org/.

[OMG03] Object Management Group (OMG): "OCL 2.0 - OMG Final Adopted Specification", 2003.

[OMG05] Object Management Group (OMG): "Unified Modeling Language: Superstructure", version 2.0, August 2005.

[PB04] S. G. Powell and K. R. Baker: *The Art of Modeling with Spreadsheets: Management Science, Spreadsheet Engineering, and Modeling Craft*, Wiley, 2004.

[Panko00] R. R. Panko: "Spreadsheet Errors: What We Know. What We Think We Can Do.", *Symp. of the European Spreadsheet Risks Intereset Group (EuSpRIG)*, 2000.

About the authors

Jan-Christopher Bals studied computer science at the University of Paderborn and the TU Berlin. His field of study is Open Source and service oriented architectures. At present he is Ph.D. student at the Software Quality Lab of the University of Paderborn.

Fabian Christ studied computer science at the University of Paderborn and the TU Berlin. His field of study is Open Source and model driven software development. At present he is Ph.D. student at the Software Quality Lab of the University of Paderborn. E-Mail: fchrist@s-lab.upb.de

Gregor Engels is in the chair of Database and Information Systems at the University of Paderborn.

Martin Erwig is associate professor at the School of EECS, Oregon State University. His research interests are programming languages and their applications.