Diploma Thesis

Management of layout information in model-driven software development processes

submitted by

Karsten Gaul
born 21.12.1982 in Eisenhüttenstadt

Technische Universität Dresden
Fakultät Informatik
Institut für Software- und Multimedia-technik
Lehrstuhl Softwaretechnologie

Supervisor: Dipl. Medieninf. Jendrik Johannes
Professor: Prof. Dr. rer. nat. habil. Uwe Aßmann

Submitted April 29, 2010
Aufgabenstellung

Management von Layoutinformationen in modellgetriebenen Softwareentwicklungsprozessen

In der modellgetriebenen Softwareentwicklung werden Layoutinformationen oft zu wenig berücksichtigt. Wenn Modelle komponiert oder transformiert werden, werden Layoutinformationen oft verworfen. Dabei tragen Layoutinformationen wesentlich dazu bei, dass Modelle von Entwicklern gelesen, verstanden und weiterentwickelt werden können.


Die Lösungen sollen in einem Prototyp auf Basis des am Lehrstuhl entwickelten Reuseware Composition Frameworks implementiert werden. Mit Hilfe des Prototypen ist eine Evaluierung der Ergebnisse durchzuführen.
## Contents

1 Introduction \hspace{1.5cm} 1

2 Reuseware Composition Framework \hspace{1.5cm} 5
   2.1 Architecture \hspace{1.5cm} 5
      2.1.1 Composition System User \hspace{1.5cm} 6
      2.1.2 Composition System Developer \hspace{1.5cm} 8
   2.2 Layout Adjustment Extension \hspace{1.5cm} 9
      2.2.1 Mental Map \hspace{1.5cm} 9
      2.2.2 Layout Extension Overview \hspace{1.5cm} 11
      2.2.3 The class CompositionDiagramUtil \hspace{1.5cm} 11
      2.2.4 Information Provider \hspace{1.5cm} 13
      2.2.5 Merger/Materializer \hspace{1.5cm} 14
      2.2.6 DiagramComposer Extension Point \hspace{1.5cm} 14
      2.2.7 CIM Example \hspace{1.5cm} 15

3 Transformation in Reuseware \hspace{1.5cm} 17
   3.1 Transformation \hspace{1.5cm} 17
   3.2 Motivating Example \hspace{1.5cm} 22

4 Modeling \hspace{1.5cm} 27
   4.1 Fragment modeling \hspace{1.5cm} 27
      4.1.1 Graphical modeling \hspace{1.5cm} 28
      4.1.2 Textual modeling \hspace{1.5cm} 29
   4.2 Diagram categories \hspace{1.5cm} 30
      4.2.1 Semantically unrestricted diagrams \hspace{1.5cm} 31
      4.2.2 Semantically restricted diagrams \hspace{1.5cm} 33

5 Architecture and Implementation \hspace{1.5cm} 35
   5.1 LaCoMe - Extending the Extension \hspace{1.5cm} 35
      5.1.1 Extension of FragmentDiagram and Bounds class \hspace{1.5cm} 36
      5.1.2 Extension of CompositionDiagramUtil \hspace{1.5cm} 39
   5.2 Implemented Components \hspace{1.5cm} 41
      5.2.1 TraceProvider \hspace{1.5cm} 42
      5.2.2 DiagramCompositionStrategy \hspace{1.5cm} 42
      5.2.3 Information Provider \hspace{1.5cm} 43
      5.2.4 Merger \hspace{1.5cm} 48
      5.2.5 ModelArranger \hspace{1.5cm} 49
      5.2.6 MultiHorizontalArranger \hspace{1.5cm} 50
IV CONTENTS

5.2.7 MultiVerticalArranger ........................................ 52
5.2.8 LayoutCompositionStrategy Extension Point ................. 55

6 Evaluation of LaCoMe ........................................... 57
   6.1 Evaluation results ............................................ 58
   6.1.1 LaCoMe manageability ...................................... 58
   6.1.2 Transformation results .................................... 60
   6.1.3 Layout adjustment approach ............................... 62
   6.2 Additional comments of the probands ..................... 67

7 Conclusion ....................................................... 71
   7.1 Related Work .................................................. 71
   7.2 Motivating Example .......................................... 72
   7.3 Diagram composition strategies ............................ 72
   7.4 Layout preservation in MDSD ............................... 73
   7.5 Future Tasks ................................................ 73

A Evaluation Sheets ............................................... 75

B Survey answers ................................................ 83
   B.1 Question 10 ................................................... 83
   B.2 Question 11 ................................................... 83
   B.3 Question 12 ................................................... 84
   B.4 Question 13 ................................................... 84
Chapter 1

Introduction

The advent of Model-Driven Software Development (MDSD) has brought up a lot of different model types for each step in the software engineering process. Each of these types has a different purpose but, in the end, all of them aim at simplifying the communication between customer and programmer as well as automatizing software production. Requirements engineering using scenarios, for example, is a widely used approach to start a software project. These scenarios can be modeled using UML’s use case diagrams or sequence charts [Gro09], they can be written in plain text as in [RW07] or in a suitable domain-specific language (DSL). Given these different options how the same data can be displayed, it is not trivial to find a tool which automatically transforms one diagram into another. For the software development process, however, transformations between different kinds of models are required. Moreover, in MDSD, the composition of models is in the focus of research.

Along with the semantic transformation and composition of models comes that of the higher level representation. A user normally does not want to understand Java or C++ but rather simple structured text or nodes and edges, i.e. diagrams. One of MDSD’s advantages is bridging the gap between programmer and user by providing a representation which both parties can understand.

At the chair of software development of the Technische Universität Dresden we developed a tool based on Invasive Software Composition (ISC) [Ass03]. This Reuseware Composition Framework [GDa, HHJZ08] provides all means necessary to compose different model components—named fragments—into one integrated model. Our previous work in the domain of diagram composition [Gau09, JG09] aims at composing layout information in a model composition environment. We implemented a framework which provides components for gathering layout information from fragment diagrams, rearranging and composing them.

One of the major tasks in our work was the arranging of the single nodes of the resulting diagram in a way that users can easily read them. Without this step there would be overlapping or even occlusion of nodes, complete rearrangement of the neighborhood relationships and other undesirable effects. In 1991 Eades et al. introduced the term of the Mental Map [ELMS91] meaning a picture created in the users mind when she navigates through and interacts with the diagram. The better this
picture is preserved throughout the whole composition process the easier comprehensible is the result. There are a number of graph drawing algorithms in the literature [DH96, HL03, MELS91] that aim at achieving the preservation of the mental map. Most of the work is based on the assumption that the diagram changes over time by adding or deleting nodes. Nonetheless, each approach starts with an automatically created layout. In this thesis, we want to propose a tool that includes the user even in this initial layout process and we assume that this layout is exactly what the user wants. This is the basis to our work of preserving as much of this initial layout as possible with some restrictions. Since there are a lot of different types of diagrams, we had to come up with a categorization regarding their layout restrictions. Additionally, we now introduce an example for the transformation process in Reuseware while our previous work only dealt with composition. This is no monolithic transformation, though, it is possible to enrich it with certain components in order to produce different results. The importance of preserving the initial layout information through the steps of transformation will be in focus here. Consequently, we are able to preserve parts of the initial layout in a whole MDSD process which results in a better readability and understanding of the different diagrams.

Furthermore, this thesis goes beyond the graphical syntax of layout languages. We now include textual modeling languages and we want to investigate whether there is some layout information to be gathered and how that can be used in transformations and compositions. More exactly we interpret a text as a sort of diagram and we want to find the restrictions of this "text diagram". With regard to the preservation of the mental map, it is inevitable to involve the customer in the diagram layout process. This, however, is impossible if there is no initial layout information to offer the user for manipulation. It will be interesting to see if there is some kind of mental map potential in text.

Another part of this work is the evaluation of our solution. Since so far there was no such test with users and we require statements about the usability of our layout composition framework, we developed an evaluation sheet which we handed out to probands. Some of these probands already used Reuseware or are even co-developers while others have no such previous knowledge. In this work, we present the results of the evaluation. Moreover, we evaluate the solution from our point of view and point out advantages and disadvantages. Finally, suggestions for improvement are discussed.

This thesis is structured as follows. Firstly, in Chapter 2, we present the Reuseware Composition Framework’s architecture as well as the layout composition extension we developed in our previous work. This chapter is meant to provide an overview of Reuseware’s functioning and the components of the layout composition extension. This implementation is later compared to the one we developed in the current work. Secondly, in Chapter 3, we introduce transformations of models as a new feature of Reuseware which is important if we want to understand the changes applied to our framework. Transformation of models has not been a feature of Reuseware yet but it is required for a whole MDSD process. Furthermore, we introduce an example of the model-driven development of a ticket shop system based on Multi-Dimensional Separation of Concerns [OT00] and the work of Roussev and Wu [RW07]. Thirdly, in Chapter 4, modeling principles are introduced. We need to distinguish graphical
and textual modeling since, in a composition and transformation environment, these cannot be dealt with equally. Additionally, the aforementioned diagram categories are introduced. In Chapter 5, we present our solutions and the new layout composition framework LaCoMe with its components explained in detail. In order to receive comments that help us improve our solution, we had to ask users to evaluate our results. The survey and the analysis of the evaluation’s outcome is presented in Chapter 6. Lastly, in Chapter 7, we refer to future work and reflect on the results and quality of LaCoMe and present related work.
Chapter 2

Reuseware Composition Framework

In this chapter we introduce the Reuseware Composition Framework (short: Reuseware). We start giving an overview about the general architecture of Reuseware followed by an overview of the layout composition extension we developed in [Gau09]. The latter so far only covers graphical layout languages and will therefore be the basis to our work where we examine the capabilities of textual languages for layout purposes. What has to be kept both for the graphical and the textual languages is the principle of the mental map which we introduce as well.

As mentioned Section 2.1 presents the overall architecture of Reuseware. After that, in Section 2.2, we introduce the layout adjustment extension and explain its components’ tasks and structure.

2.1 Architecture

At the core of Reuseware is CoCoNut, a framework which provides the composition technique. Furthermore, mechanisms for the integration of Reuseware into other technologies are offered. This customary extension option is managed via extension points.

Basically, there are two user roles in Reuseware; the Composition System Developer and the Composition System User. The first is for defining the composition system and the second is responsible for using it. In the architecture these roles reflect in the Reuseware metamodels which are depicted in Figure 2.1. For the developer there are CompositionSystem and ReuseExtension. The user, on the other hand, requires CompositionProgram and Fragment.

As depicted in Figure 2.1 ReuseExtension has a reference to Ecore, the metamodel EMF [Foua] models conform to. Ecore is based on the Essential Metaobject Facility (EMOF), a reduced MOF 2.0 [Groa]. EMOF, although just a reduced version, is the better choice here since it concentrates on the most important elements and mostly is compatible to EMF.

1The metamodels are created in Fujaba (http://www.fujaba.de/)
2.1.1 Composition System User

The previously defined composition systems are used by Composition System Users. For that, users define fragments—the components in Reuseware—which they can compose. Accessing or modifying the fragments is possible via their composition interface. The concepts for these interfaces are modeled in the Fragment metamodel.

Furthermore, users define CompositionPrograms which specify the composition of the involved fragments. What is especially important for this work is the difference between physical and derived composition program. The first is created manually by the user choosing fragments from the fragment repository and adding them to the composition program. Using this option implies the employment of the generic composition language of Reuseware, which is defined in the CompositionProgram metamodel. There is, however, yet another possibility. The second, as we will discuss later in Section 2.1.2, is created semi-automatically by defining composition languages and deriving the composition program by following the concepts of these languages.

**Fragment**

Figure 2.2 shows the Fragment metamodel in detail. As mentioned, the components in Reuseware are called fragments whereof each has a composition interface (cf. Figure 2.2(b)). These interfaces consist of a set of Ports that are connected to model elements. Through these interfaces fragments can then be accessed and modified. Elements that are not connected to ports cannot be modified in a composition and are therefore hidden.

As depicted in Figure 2.2(a) there is a link to the Ecore metamodel to provide genericness with regard to the modeling language where EObject is some arbitrary element and EStructuralFeature describes a reference between model elements. Accessible elements within a model are called ReferencePoint. Prototype, Anchor and ValuePrototype are such reference points. VariationPoints—Hook, Slot and ValueHook—are those elements that can be modified. When a prototype is bound
to a hook, itself and its children are copied and put into the place where the hook
is. Binding an anchor to a slot causes references from the slot be redirected to the
anchor in a composition. Value prototype and hook prototype composition is quite
similar to the prototype-hook binding but for the case that attributes are copied.

CompositionProgram

FragmentInstance, PortInstance, CompositionLink and Setting are the concepts
of this metamodel. FragmentInstance is the concept that describes a model frag-
ment in a composition and it always exists in the context of a composition program.
There can be different fragment instances of the same fragment in one or in different
composition programs. Different instances of the same fragment are distinguished by
their names when they are in the same composition program.

Between a fragment and its instance there is always a connection which can be
checked by the method fFragment(). An error is raised when there is no fragment
with the given ID in the repository.
For defining compositions the CompositionLink concept is required. Such links connect PortInstances. These PortInstances can be derived automatically too when a FragmentInstance is valid. Whether a connection is valid is checked by a match operation.

### 2.1.2 Composition System Developer

Often, required composition functionality is missing in existing languages. Therefore, we require a mechanism to extend existing languages in a way that there are no conflicts with previously developed software but that these extensions enable us to use these languages in our reuse and composition environment. Reuseware gives a developer the tools to create composition systems for a given language. The concepts of CompositionSystems have to be developed as well as the rules which determine, where those concepts are found in the composition language. ReuseExtension is that language in Reuseware. Both, CompositionSystem and ReuseExtension have a EMFText [GDb] syntax. Additionally, ReuseExtension uses OCL [Grob] for expressions.

#### CompositionSystem

A Composition System Developer uses the Composition System Specification (CSYS) language to define concepts. These concepts are the basis for component model and composition language. They are semantically related to the operations of Fragment and CompositionProgram. The metamodel of CSYS is the basis for a specification language for composition systems. In turn, specifications of that language are the basis for reuse extensions which allow for the introduction of composition concepts into existing modeling languages.

Composition systems consist of FragmentRoles which basically are component types. Each component (fragment) in a composition system—i.e. advices and core components in aspect systems [KIL+97]—have a certain role. FragmentRoles again, consist of a set of PortTypes.

In addition to fragment roles, composition systems consist of CompositionAssociations which can be either Configurations or Contributions. We can define here whether links between two port types are allowed and whether they are contributing or configuring. A contribution triggers the copy of a fragment while the configuration reuses an existing fragment.

#### ReuseExtension

Along with a EMFText syntax, the ReuseExtension metamodel forms the reuse extension language of Reuseware. The Reuse Extension Specification (REX) combines composition system and a concrete modeling language by defining how reuse concepts can be found in that modeling language. REX consists of two parts whereof one is used for the specification of component models and the other for the specification of composition languages.
Specifying a component model means the developer defines which elements of a model have which variability types and in which ports they are grouped.

On the other hand, specifying a composition language means giving composition semantics to an existing modeling language. This language can then be used to define composition programs without using the generic language of Reuseware. Introducing this new language means, that more details of Reuseware are hidden from the user. This is an important point for our current work since we make use of that option.

In order to be able to specify conditions under which a model element has a certain variability type an expression language which allows for the creation of such rules over Ecore metamodels is required. It was decided to use OCL since it allows for defining conditions and queries for Ecore models and can be used directly in an EMF-based tool.

### 2.2 Layout Adjustment Extension

After having introduced the basic architecture of the Reuseware Composition Framework, we now have a look at the layout composition framework we developed in [Gau09, JG09].

This layout composition is an extension of the Reuseware Composition Framework aiming at the user-centered design of diagrams which are the outcome of a composition. In order to be user-centered, a diagram has to obey some rules which are summed up by the term of the Mental Map [ELMS91]. We assume that users make up their minds about where to put an element of a diagram before they move it and we provide means to preserve the layout the user created and manifested in her head, i.e. the mental map.

#### 2.2.1 Mental Map

There are different opinions about which goals have to be met in order to produce the best diagram layouts (cf. [DH96]). It should be difficult to meet all goals that are considered helpful to preserve the mental map. In his master thesis, Coleman [Col93, p. 8-10] gives a partial list of these so called aesthetics. He mentions that vertices, i.e. nodes, should not be placed too close together, that the width of the placement should be minimized and that the placement of existing vertices and edges should change as little as possible when changes are made. As stated, not all of these aesthetics can be handled by only one algorithm but neither is it sufficient to stick to just one criterion like the node movement. Saffrey and Purchase [SP08] found that preserving the mental map is no trivial task. Only fixing node positions in a changing diagram can cause problems in understanding it too, since this can lead to overlaps. In summary, different goals are likely to cause conflicts among each other.
Due to this we chose three goals we think fit best in order to preserve the mental map and are least conflicting. These goals are the following:

**Goal 1:** disjointness of nodes

**Goal 2:** keep the neighborhood relationship of the nodes

**Goal 3:** compact design

As mentioned above, these goals can already be found in Coleman’s master thesis. Although the first goal—disjointness of nodes—has the biggest impact, it is not even close to be sufficient if one wants to create a layout that improves the comprehensibility of a diagram. When there are labeled nodes in a modeling process, each representing a chunk of software that is to be composed to a bigger system, these nodes should not loose the relationship to their neighbors. In [Gau09] we mapped this concept of software composition to cartography. When road maps are extended because a new village was founded, this village has to be inserted at its appropriate position and maybe the map has to be rescaled but all the cities and villages around it have to keep their neighborhood relationship. The third and last goal is compact design. Again we can see the importance of the composition of goals and their weight. The most compact layout clearly is the one where all nodes are stacked. This, however, does not comply with the readability guidelines of diagrams. Applying the first
two goals beforehand though will result in a good layout.

We required some algorithm which enabled us to not only create some design which was widely accepted but one which can be adapted to the users needs. More precisely, we required an additional step in the process of layout creation. Misue et al. called this step layout adjustment [MELS91] and they proposed some algorithms to perform this adjustment. Uniform Scaling and Horizontal Sorting are just two of the great variety of layout algorithms that can be applied in this step.

2.2.2 Layout Extension Overview

Since we wanted our framework to be generic, we had to invent a component-based architecture whereof few components have to be exchanged in case of changing the fragment layout language for example. Our framework consists of the five components depicted in Figure 2.3, each required for specific tasks within the process. Firstly, layout information has to be gathered both of the involved fragments as well as of the composition script. Therefore, Source Information Provider (SIP) and Target Information Provider (TIP) have been implemented. When this task is completed, the composition order has to be determined. This is crucial for some layout algorithms because a non-deterministic order would yield different results each time the algorithm is applied. We provide the Comparator for this task. When this is done the fragments are arranged according to the specific layout adjustment algorithms by our Arranger. The final step of the composition is the materializing of the fragments in a single diagram and this is done by our Materializer. There is detailed information about these components in [Gau09, JG09].

2.2.3 The class CompositionDiagramUtil

Only having the components of the layout composition framework would not be sufficient. We had to implement at least one central class which is responsible for the orchestration of the actions that need to be performed in that composition process. Actions like the initialization of information providers and the execution of Comparators require a certain order because Comparators have nothing to compare if no layout information was gathered before. The class CompositionDiagramUtil (CDU) is the conductor for our framework and the orchestration is done in its composeDiagrams method.

Firstly, the extension points had to be initialized so that the components of our framework are known to the different methods. After that, both the TargetDiagramInformationProvider and the SourceDiagramInformationProvider are called and the layout information is assigned to the model elements of receiving and contributing diagrams. The third step is the call of the Arrangers. Since we have two kinds of Arrangers—Multi- and SingleSourceDiagramArranger—we had to check for them and handle the method calls differently. MultiSourceDiagramArrangers are called and, when they are finished with the layout adjustment, the resulting diagram is merged. On the other hand, SingleSourceDiagramArrangers require a merge after one arrangement step.
CHAPTER 2. REUSEWARE COMPOSITION FRAMEWORK

Figure 2.4 presents the tasks, CompositionDiagramUtil has to perform. On the left hand side, the initialization of the extension points is depicted. Source- (SIP) and TargetInformationProvider (TIP) are initialized as well as the Comparator (COM) and the Composer (CMP)\(^2\). On the right hand side of Figure 2.4 the required tasks for the layout adjustment are displayed. The order of these actions is set by CDU within its composeDiagrams method.

Figure 2.5: The parts of the diagram composition. (a) source and target information (b) composed diagram

\(^2\)The Composer encapsulates Arranger and Merger/Materializer
2.2. INFORMATION PROVIDER

We have to have a closer look at the exchangeable parts of this architecture, the information providers and the materializer. In our previous work we discussed examples where both the fragment layout language and the composition layout language had a graphical syntax, i.e. GMF [Foub] or TOPCASED [Val] were used as layout languages. Since we did not want to restrict the user in her choice of languages, the components that work directly with these languages had to be adaptable. Thus, if any new layout language comes into play—be it on the fragment developer or on the fragment user side—at least one information provider and the materializer have to be exchanged. Figure 2.5 depicts this composition, involving different layout languages. As is shown in Figure 2.5(a) the source and target diagram information stems from the involved fragments and the composition script. This information can be of different type since, as mentioned, the fragment layout language does not need to be the one that is used for the composition script. After the whole arranging, the materializer creates a new diagram in a specific layout language which is depicted in Figure 2.5(b).

**SourceDiagramInformationProvider.** When fragments are laid out on a canvas they have certain bounds (or layout constraints). These layout constraints are of importance once we want to compose different fragments and thus we have to gather this information. We think that the creator of such fragment diagrams does not make up her mind about the x and y values of the overall diagram but just puts it wherever she likes. Assuming this information redundant since the reuse of fragments in different composition programs requires different information about the placement of each node, we only gather width and height.

**TargetDiagramInformationProvider.** As mentioned, Reuseware has its own generic composition language which has a graphical GMF syntax. While laying out the composition program’s diagram, fragment users make up their minds about how the composed diagram should look like. Since it is our task to preserve this layout, we have to gather the x and y position of the fragment diagram’s illustrations. However, width and height of these illustrations are not important.

![Figure 2.6: Difference between FragmentDiagrams (background) and their illustrations in the composition program](image)

Figure 2.6 shows why we have to gather the different information from different sources. The illustrations in a composition program may be smaller or bigger than the actual fragment diagrams which is why their width and height should not be regarded during the adjustment process. On the other hand, we want to move the original fragment diagrams to the x and y positions of their illustrations. Therefore, we do not have any use for the x and y information of the original.
2.2.5 Merger/Materializer

There is one kind of component we named Merger in [Gau09] and Materializer in [JG09]. The notion of materializing the previously applied changes made us rename it for the OOPSLA workshop paper [JG09]. We stick with the Merger suffix for this thesis though since merging the gathered information of the involved fragment diagrams is a understandable notion as well.

Although our Arrangers already reset the layout information this did not have any impact on the layout of the diagram so far. We just set the Bounds — i.e. x, y, width and height — internally. Given that information however, we are now able to apply it to the diagrams using layout language dependent Mergers. The canMerge method checks for the applicability.

```java
public boolean canMerge(FragmentDiagram receivingDiagram) {
    if (!receivingDiagram.getDiagramRoots().isEmpty()) {
        return receivingDiagram.getDiagramRoots().get(0)
            instanceof Diagram;
    }
    return false;
}
```

Listing 2.1: canMerge(FragmentDiagram) of GMFMerger

Listing 2.1 shows the GMF version of this method. First, in Line 3, the receiving diagram is checked for content. If it is empty, no merge has to be performed. This method returns true if the first element is of type org.eclipse.gmf.runtime.notation.Diagram. After that, the method merge is executed which then calls layout language specific rearrangement methods.

2.2.6 DiagramComposer Extension Point

In order to enable users to choose freely which components they want to employ, we implemented the diagramComposer extension point. This extension point must have a XML schema definition (EXSD) [Con].

```xml
<composer
    id="id"
    comparator="org.example.DiagramComparator"
    arranger="org.example.ExampleArranger"
    merger="org.example.LanguageMerger"/>
```

Listing 2.2: DiagramComposer

One valid instance of that schema is shown in Listing 2.2. The id is a required attribute while all the other can be left out. What we did not need to specify here is the employment of the information providers. These are executed automatically. Not having specified a comparator can yield wrong results since Reuseware has no
2.2. LAYOUT ADJUSTMENT EXTENSION

deterministic algorithm of choosing the next fragment to compose. On the model level this is not important. However, on the level of diagrams, we need to have a certain and moreover, a deterministic order of composition. Otherwise we will have a different result each time we execute the composition process. However, it can be left out all the same. An **arranger** is the important component for layout adjustment which makes it almost indispensable. Even that one can be left out though. The last attribute is the **merger** without which there will be no application of the changed layout information to the diagram. Although we thought about the combination of Arrangers in that work, we did not find any use case for it and thus we did not implement this option. Therefore, there can be only one of each of the attributes.

2.2.7 CIM Example

![Figure 2.7: (a) Composition program (b) Composed diagram with HorizontalArranger applied](image)

An example taken from [Gau09] is depicted in Figure 2.7. We present a composition program in Figure 2.7(a) and the result with additional layout adjustment in Figure 2.7(b). For the composition program the generic composition language of Reuseware with its GMF syntax was used. The named rectangles illustrate fragment diagrams which the user adds from the fragment repository in order to compose them. Gray rectangles are those specified to be receiving fragments in the composition process while the white ones are contributing fragments. The receiving fragment is the one into which the result is composed. In our case, this one is empty. Filled circles symbolize **contributing** ports which can be connected to a **receiving** port (empty circle) via composition links. Dotted circles are yet another type of ports for the configuration of involved fragments. These **configuring** ports are also connected by composition links.

This example is partially composed of fragments which themselves are results of a composition. Illustrated by only one rectangle in the composition program, the **Ethernet IP Interface** originally consists of the fragments **IP** and **Built-in Ethernet Hub**. The **Core** is an empty fragment which is the target for this composition. Simply composing the components according to the composition program would yield a diagram where all nodes are at the position (0,0). This, however, destroys the
mental map the user had while arranging the composition program. Even providing the option to arrange the diagram strictly according to the user's arrangement does not produce a satisfying result since, as can be seen in the exemplary composition, the fragments would overlap. **ADSL Static IP Interface** would cover parts of **Ethernet IP Interface**. The reason for that is the different size of the fragments and their illustrations. Thus, we have to apply the layout adjustment which is meant to keep as much information from the composition program as possible while rearranging the result according to the three goals of the mental map concept.

There are several layout adjustment algorithms in the literature [ELMS91, MELS91, MSTH03] and in [Gau09] we implemented *HorizontalArranger* and *UniformScaling* as examples. Figure 2.7(b) depicts the application of the *HorizontalArranger* which checks for overlaps and moves the overlapping fragment in x direction until the overlap is eliminated. We noticed, however, that a Comparator is required which is responsible to choose the next fragment for the composition. We have to start at x=0 and traverse the diagram looking for the lowest x value. Chapter 6 in [Gau09] evaluates the adjustment algorithms and although there are some disadvantages we think to have proven the possibility to manage layout adjustment processes in a composition environment. The three goals for preserving the mental map can be met provided the adjustment algorithm can be fed with the parameters from the composition program.
Chapter 3

Transformation in Reuseware

Reuseware not only covers the composition of fragments but it can also be used to transform models from one representation into another within a model-driven software engineering process. In the previous chapter we presented the architecture of Reuseware and that of the layout composition extension and the notion of the mental map. In this chapter we discuss the terms *transformation* and *composition* in more detail.

Transformation and composition are two sorts of dealing with components. Transformation mostly keeps the amount of data to be displayed and changes the view the user has on that information. This is achieved using some templates according to which one representation can be transformed into another. Composition on the other hand, is performed on the same view on the data. While the representation is kept, the amount of data changes in a composition. In Reuseware, however, these two concepts are combined. This combination has several advantages which are discussed in this chapter. One of these advantages is the gain of flexibility in a MDSD process.

At first, that is in Section 3.1, the transformation process is discussed and we point out what is important for our aims. After that, in Section 3.2, the ticket shop example is introduced which uses this new concept of Reuseware. This thesis was basically inspired by the requirements of that ticket shop and we present our research results using diagrams and model transformations of this example.

3.1 Transformation

According to [MCG05] there are different kinds of transformations. An *exogenous* transformation, for example, is a transformation between models written in different modeling languages while, on the other hand, there is no change of modeling languages in an *endogenous* transformation. Furthermore, the kind of transformation can be divided into *vertical* and *horizontal* whereof the first is used to add details to or remove them from a model. Therefore, source and target models reside on different abstraction layers. The latter, i.e. the horizontal transformation, is not used for such a refinement but both models reside on the same layer of abstraction like it is the case for refactoring [MMBJ09].
CHAPTER 3. TRANSFORMATION IN REUSEWARE

<table>
<thead>
<tr>
<th></th>
<th>horizontal</th>
<th>vertical</th>
</tr>
</thead>
<tbody>
<tr>
<td>endogenous</td>
<td>model composition [Gau09]</td>
<td>simplification [Wir71]</td>
</tr>
<tr>
<td>exogenous</td>
<td>language migration</td>
<td>code generation</td>
</tr>
</tbody>
</table>

Table 3.1: Transformation dimensions from [MCG05]

Table 3.1 shows the dimensions of transformations and examples for the combinations. An example for the endogenous and horizontal transformation is the GREAT [Chr05] tooling which allows for refactoring of UML class diagrams. Reuseware, however, bridges the gap between code (or other model) generation (exogenous/vertical) and model composition (endogenous/horizontal). Furthermore, Reuseware can transform not only UML class diagrams but different models.

When software is created according to well known approaches like the Rational Unified Process (RUP) ([IB] or the "V-Model XT" [IAB] many different models are used in the single steps. Changing the type of model, however, does not mean to discard the layout information a model contained. In most cases, the representation of data is changed while the amount of data is either kept or even enlarged (exogenous/vertical transformation). Transforming one type of model into another is a time consuming task if performed manually. Therefore, a lot of tools are on the market that provide an automatic transformation of the semantic model.

![Figure 3.1: (a) Monolithic transformation (b) transformation in Reuseware](image)

Usually, in MDSD processes we do not have access to the internals of a transformation but the source model is transformed and integrated into the more concrete model. This approach however, is a drawback since in many cases design flaws are detected in a software development process. Often, these flaws are detected in later stages of the process which requires access to the design components in order to fix the software.

Another possible influence on the software project are customer requirements. Those are likely to change over and over again and the software has to be adapted.
In a monolithic transformation process, this is hardly possible.

As depicted in Figure 3.1(a) a monolithic transformation process is straightforward. The source model is traversed and scanned for certain patterns. These patterns conform to a transformation rule according to which they are transformed. Kleppe et al. [KWB03] defined a transformation that way.

With Reuseware, on the other hand, users are able to find the fragments involved in the transformation they would like to change. The reason for that advantage is depicted in Figure 3.1(b). If we want to understand that approach, however, we first have to look at the requirements of a composition system which are shown in Figure 3.2.

For a composition system three things are required.

A **Composition Technique** specifies how components are physically composed. In Reuseware this technique is based on *Invasive Software Composition* [Ass03] and is therefore an invasive software composition technique. This means that components are merged using graph rewriting.

Using a **Composition Language** enables the user to specify composition programs which then define concrete compositions. Reuseware’s composition language allows for instantiating fragments into fragment instances and for connecting them by composition links. For the latter, it is first checked whether they can be merged. This is done by a match.

The third element is the **Component Model** which determines the composition interface of components. Such a composition interface is important for the instantiation of components. After having them instantiated, the components can be used in a concrete composition program.

As depicted in Figure 3.1(b) the transformation in Reuseware is not a monolithic transformation. A composition system which uses fragments—in Reuseware, this is the expression for components—to compose models is employed. From the document of which a transformation is required, a composition program is derived. Afterward, this composition program is enriched by additional information required to get a more
concrete document. Hence, Reuseware employs a combination of an exogenous/vertical transformation between the two languages and an endogenous/horizontal transformation for the composition of the enriching fragments. In our example we start off with an OpenOffice document (ODT) where some text is marked with a special text type.

Figure 3.3: BookTicket.odt

Figure 3.3 shows the file BookTicket.odt and the above mentioned special text type the Actors are marked with. In this textual form, however, we can only recognize them because of the bold face. Those three actors are the actors of a use case for the exemplary MDSD process presented in Section 3.2. Now the use case description had to be refined and therefore this OpenOffice document is transformed into a document in another language. For this, a composition program is derived. The program Main.fc for this example is shown in Figure 3.4. Since it is a derived composition program it is laid out automatically. This will be one point of discussion when we come to preserving the mental map of the original diagram.

Figure 3.4: Automatically derived composition program Main.fc

In [Gau09] we explained how the composition diagram helps composing the fragments the user chooses. The MDSD process does not allow such an interactive composition. This, however, is the only thing that changed. The composition program still represents the fragments to compose and in the motivating example there is an empty target into which the fragments are composed. These source fragments have to be developed by the fragment developer. Fragment users have to know the names of these fragments since the actors of the OpenOffice document have to match these names. Otherwise the composition program would not find the fragments and the composition would fail. Apart from this, the composition is quite the same as the one presented in [Gau09].
3.1. TRANSFORMATION

In Figure 3.5 we annotated Figure 3.1(b) with the requirements of a composition system. As mentioned, the composition language is used to define composition programs. Thus, in our example the composition language is OpenOffice. Another requirement was the component model which is UseCaseInvariant in this transformation. Therefore, the fragment developer has to define the fragment’s composition interfaces in that domain specific language. Invasive software composition is then used to compose the different fragments into a target fragment. Actually, this is just the first stage of the software development process. As stated, we employ six languages in the transformations. In order to transform documents written in one language into documents written in another language, a different component model has to be developed and fragments in the target language have to be created. Additionally, if the result is to be refined—we want to get executable code in the end—the composition program has to be enriched with the required information. Therefore, component models in UseCaseInvariant, UML Class, ValueFlow and Java were defined.

There is one important thing about Figure 3.5 we did not discuss yet. As presented in [Gau09] the target information for our layout composition stems from the composition diagram. Now, however, we have no influence on this layout anymore because it is created automatically. We have to gather some constraints the user can manipulate, though. Without that precondition, our whole approach regarding the preservation of the mental map is not realizable. Since the user writes her use cases in OpenOffice documents we have to use that. Fortunately, there is a back link from the composition program to the OpenOffice document (derivedFrom). This provides us all the access we require in order to get layout information since we can get the original file—in a composition, Reuseware only works with copies—and there we find the layout information. How we managed to get that and what kind information we extracted will be discussed in Section 5.2.
3.2 Motivating Example

In this section we introduce the example that inspired us to work on the topic of maintaining layout information in a transformation process with Reuseware. It is based on the example presented in [Joh08]. The resulting software allows users to order tickets as well as change seats and other related actions.

![Figure 3.6: (a) Transformation (b) Enrichment](image)

As mentioned, transformations are widely used in the software development process to get from a specification to executable software. In the case of this example, some use cases were defined which are then transformed stepwise to Java code. In this MDSD process six languages are employed. For further information about these languages, please refer to [Joh08] and the website of this example\(^1\).

**OpenOffice:** On the most abstract level OpenOffice documents are utilized for specifying use cases.

**UseCaseInvariant:** Each actor gives and takes values in a running system but the overall amount of these values has to stay invariant. A modeling language based on an Ecore metamodel and a textual EMFText syntax was defined in [Joh08].

**UML Class:** In order to automatically obtain a data model UML class models are used on the next abstraction layer. Therefore, the Ecore metamodel of the Eclipse UML2 project is required as well as the TOPCASED editor.

**ValueFlow:** The ValueFlow metamodel for defining the order of the passing and receiving of values within the system was developed.

**SecProp:** The SecProp language was developed to introduce security properties. This language again has a GMF syntax.

**Java:** On the most concrete layer of abstraction Java is used. The Ecore metamodel is defined by JaMoPP [GDe] including the required tooling to handle Java models. With JaMoPP the gap between modeling and Java was closed.

\(^1\)http://reuseware.org/index.php/MDSD
3.2. MOTIVATING EXAMPLE

As shown in Section 3.1 the Reuseware Composition Framework has its own approach to transformations. In Figure 3.6 this approach is depicted in a slightly different way. On the left hand side the transformation process is shown with the composition language on top, the derived composition program in the middle, and the composed or/and transformed fragment diagram at the bottom. On the right hand side, i.e. (b), the enrichment of models by composing fragments with the composition program is shown. These enriching fragment diagrams, however, can be developed in different languages depending on which result is desired.

Figure 3.7: (a) UseCase composition (b) Participant and UseCase Composition

This example follows the concepts of a Multi-Dimensional Separation of Concerns [OT00]. There are different concern dimensions like the usecase or the participation dimension (cf. Figure 3.7). A single concern in the usecase dimension is the BookTicket depicted in Figure 3.7(a). Customer, Clerk, and Bank depicted in Figure 3.7(b) are concerns of the participation dimension. By enriching the composition diagram with different sources—in different languages—the view on the result changes.

```plaintext
fragment role UseCase {
    port Name {
        UseCase.name is value hook {
            point expr = $'name'$
        }
    }
}
```

Listing 3.1: fragment role 'UseCase' definition

Figure 3.7 shows the choosing of different sources which is managed by an activator file in this example. Only activating the usecase variant yields the result depicted in Figure 3.7(a). The composition language is the OpenOffice document, the BookTicket.odt, and the fragment is modeled in UML. The fragment diagram is an empty UML use case diagram. As explained in 2.1 there is a ReuseExtension language (REX) that connects the concrete modeling language to the composition system. For this specific example the usecase.uml.rex is used.
Listing 3.1 shows this file. In it we see the specification of how the concepts of the composition can be found in the modeling language. The FragmentRole UseCase is introduced which has a Port specification called Name. In Line 3 is the definition of a ValueHook. It says that the attribute name of a UseCase is such a ValueHook and can be replaced. There is one such rex file for the composition language as well which defines a port of the same name. However, in this definition, the expression 'name' gets a value. This combination yields the use case diagram on the right hand side of Figure 3.7. The placeholder NAME in the fragment diagram which enriches the composition program is replaced by the actual name of the use case BookTicket defined by its file name.

Figure 3.7(b) depicts the transformation of the OpenOffice document enriched not only with the use case placeholder but with actors involved in a use case; so called Participants. In this example the actors Customer, Clerk and Bank participate. Again there is a rex file for the participation concern which determines how the actor in the component model is handled in a composition. This concept has to be applied three times in the composition process. Once for every participating actor of the use case. The result on the right hand side of Figure 3.7(b) shows the three actors each with a participation link to the use case BookTicket.

| 1 | fragment role Participant { |
| 2 | odfText::SpanType if $styleName = 'Actor'$ { |
| 3 | fragment expr = $'Participant:'.concat(mixed->at(1).getValue()) |
| 4 | .oclAsType(String)).concat('_').concat(ufi.trimExtension()) |
| 5 | .segment(-1)) |
| 6 | ufi expr = $Sequence{'org','reuseware','lib','systems', |
| 7 | 'participation','lib','Participant'.concat(variant)}$ |
| 8 | port Name { |
| 9 | $'name' = $mixed->at(1).getValue()$ |
| 10 | } |
| 11 | } |
| 12 | } |
| 13 | association Participation { |
| 14 | odfText::SpanType if $styleName = 'Actor'$ { |
| 15 | fragment expr = $'Participant:'.concat(mixed->at(1).getValue()) |
| 16 | .oclAsType(String)).concat('_').concat(ufi.trimExtension()) |
| 17 | .segment(-1)) |
| 18 | port expr = $'Contrib'$ |
| 19 | --> |
| 20 | fragment expr = $'UseCase:'.concat(ufi.trimExtension()) |
| 21 | .segment(-1)) |
| 22 | port expr = $'Rec'$ |
| 23 | } |
| 24 | } |

Listing 3.2: odt.rex for the composition language
3.2. MOTIVATING EXAMPLE

Listing 3.2 shows the odt.rex file. In Line 2 it is specified that only those text elements of the OpenOffice document are participants of our use case that are of type SpanType and have the style name Actor. For the Participation association these actors are the contributors while a UseCase is the receiver. The composition program depicted in Figure 3.4 shows the composition program participation concern of the three actors specified in the OpenOffice document. The naming schemes for fragments and ports defined in Listing 3.2 can be found there.

As mentioned in Section 3.1 the composition language can be exchanged. For some concerns of this example it has been done. One of these concerns was the introduction of counter actors which are not defined in OpenOffice documents but in the UseCaseInvarient files. Thus, a rex file for the UCINV composition language had to be created. The resulting composition program, although enriched with the same fragment diagrams, now yields a different result since the information that is provided by the composition language is different.

Figure 3.8 displays this additional concern. The file BookTicket.ucinv introduces the counter actor Hall and not activating the OpenOffice participation concern yields the fragment diagram shown in Figure 3.8(b).

```plaintext
fragment role Participant {
  CounterActor {
    // to add counter actors
    fragment expr = $'Participant':concat(name).concat(' ').
      concat(ufi.trimExtension()).segment(-1)$;
    ufi expr = $Sequence{'org','reuseware','lib','systems',
      'participation','lib','Participant'.concat(variant)}$
    port Name {
      $'name'$ = $name$
    }
  }
}

association Participation {
  CounterActor {
    fragment expr = $'Participant':concat(name).concat(' ').
      concat(ufi.trimExtension()).segment(-1)$;
    port expr = $'Contrib'$
    -->
    fragment expr = $'UseCase':concat(ufi.trimExtension())$.
  }
}
```

Figure 3.8: (a) BookTicket.ucinv (b) Participant and UseCase Composition
segment (-1)$
port expr = '$Rec$
}
}

Listing 3.3: ucinv.rex for the composition language

In Listing 3.3 the ucinv.rex file is shown. Although Hall is a counter actor in the composition language it is a normal participant like Customer, Clerk or Bank in the results. The definitions of fragment role and association are equal to those of the OpenOffice document rex. Similar to the SpanTypes, in Line 2 it is specified that only those elements play the Participant role that are of the type CounterActor. The definition of the association also handles only the elements of type CounterActor. In Lines 15-21 the specification of Participation is shown. It again consists of a Participant fragment (Lines 15-16) which has a contributing port, an association arrow -->, and a UseCase fragment that provides the receiving port.

One important thing about this example is the clear separation of concerns. In the process of software development customers often change their minds about the necessity of requirements. They either want them removed or changed or they come up with completely new requirements the system was not designed for in the first place. In this example it is shown how developers can get access to basic artifacts throughout the whole development process. This enables them to work with a consistent system at any point in time. In the literature this concept is referred to as Separation of Concerns (SoC) [OT00] and is required to omit tangling and scattering of code in a software system.

Being our motivation for this work, we derived the requirements for the improvement of our layout composition framework from the transformations and diagram types of this example. As a start, in Section 4.2 we use it to categorize diagrams according to their layout restrictions. Afterward, in Chapter 5 we present how we managed to improve our framework in order to preserve layout information for this motivating example with regard to the categorization we developed. We present the required components and discuss their contribution to our aims. In Chapter 6 we present the evaluation we did in order to receive comments and recommendations about the usability from users.
Chapter 4

Modeling

Until now we have introduced the concepts of the mental map, transformation, composition and the architecture of our layout extension to the Reuseware Composition Framework presented in [Gau09]. In this chapter we want to discuss the sources we get the layout information from already and the sources that have yet to be exploited. In other words, we want to present the graphical modeling and the textual modeling which we already referred to in previous chapters. This will provide the knowledge necessary to adapt the TIPs and SIPs of the layout extension of Reuseware (cf. Section 2.2).

In our previous work (cf. [Gau09, JG09]) we discovered some drawbacks in the generality of our extension framework. This was based on the different categories of diagram layouts that are important in software engineering. On the one hand there are diagrams of which the nodes can be arranged freely on the editor’s canvas. In this context we have to recall that edge crossings are not in the set of goals for preserving the mental map we listed in Section 2.2. On the other hand there are diagrams that are restricted in the way their nodes can be laid out. In order to improve our previous work we discuss a categorization of diagrams that we had to come up with in this chapter as well.

In Section 4.1 we present both the graphical and the textual modeling. Commonalities as well as differences are pointed out with regard to the gathering of layout information. The next section, i.e. Section 4.2, is about a categorization of diagram layouts.

4.1 Fragment modeling

When components are to be modeled we are given at least two options of performing that task: we can either use a layout language which provides an abstraction of the model, i.e. a diagram, and arranging the diagrams’ elements or we can write text.

Figure 4.1 depicts the difference between the actual model and the layout language. The syntax of the layout is an abstraction of a model and does therefore not influence the semantics of a model. Changing a textual model document however (i.e. with
EMFText [HJK+09], changes the model itself. This coupling of representation and model requires special layout management approaches.

### 4.1.1 Graphical modeling

Modeling with graphical representations is a straightforward task. Due to the fact that the diagrams are laid out on an editor canvas and serialized in a XMI [Groc] file we can gather the required information from that source directly.

Figure 4.2: A diagram of a model

Figure 4.2 depicts a fragment called **Built-In Ethernet Hub** from [Gau09] with a GMF representation. From this diagram we can gather layout information like the width and height of such a fragment.

In Figure 4.3 an exemplary composition program from [Gau09] is shown. The values for x and y for the composition of the involved fragments are in the drawn diagram. This is one of the advantages of model-driven software development; even
users that are not experienced in modeling understand the representations. Thus, they can learn quickly how to intervene in the layout process, provided they are given this option.

4.1.2 Textual modeling

In the previous subsection we discussed the graphical modeling and where to find the layout information there. However, models cannot only be developed graphically but also textually. To some this is more handy than drawing rectangles or lines on a canvas. People that are familiar with programming languages, for example. Furthermore, in a MDSD process at least the last stage of development often is a textual language like Java because the aim of such a process is to get executable code.

Having a modeling language with a sophisticated textual syntax sometimes is as helpful as a graphical representation. Brackets, lines, columns and even whitespace are only some of the variety of syntax properties that can be employed to create a model that can be read by experienced users.

However, for our purposes, i.e. the transformation of diagrams, it is quite difficult to employ textual languages. If we, for example, intend to transform a textual model into one with a graphical representation it is hard to find layout information we could use in the resulting diagram. Therefore, the management of layout information is harder if there is a difference in the representation of models. It gets even harder if the semantic restrictions come into play. In textual languages it is not always possible to keep a certain arrangement of elements since the result would not be semantically correct. An example in Java would be that a variable cannot be used before it was defined.
4.2 Diagram categories

In this section, we present some diagram types important for software modeling. We point out similarities as well as differences of the diagram types and consequentially come up with a categorization. The Unified Modeling Language (UML), for example, has 14 diagram types which can be divided into two categories. These categories are listed in the superstructure specification [Gro09] and refer to the uses these diagrams have been invented for. On the one hand, there are the structure diagrams like class diagrams or component diagrams. These specify the things that have to be in the system being modeled. On the other hand, there are the behavior diagrams for modeling the behavior of the system.

Figure 4.4: Constellation Diagram with o restriction

The categorization of diagrams the UML proposes is one way to do it. We, however, are not interested in what the diagrams represent but how they are laid out. Since we have to check diagrams for their syntactical similarities in order to transform one into the other or compose them, we require a different categorization. A coarse-grained categorization is the one between semantically restricted and semantically unrestricted diagrams. The latter can then even be subdivided into more fine-grained categories. There are those diagrams that can be laid out freely on the canvas like the CIM diagrams [Dis08] we used in our previous work [Gau09, JG09]. And on the other hand, there are diagrams like UML sequence diagrams where the nodes have to be at \( y = 0 \), life lines are parallel to the x-axis and messages are horizontal lines.

In Table 4.1 we present a list of diagrams and the attributes which may only change to a certain degree. The class diagram, for example, has no checkmarks in this table because—although it is not always wise—we can put its nodes on the canvas where we want them. The sequence diagram, however, is strongly restricted regarding the x and y values of its nodes. As mentioned, the nodes must be laid out at \( y = 0 \) and they should be ordered in a way that the whole diagram can be read from left to right along the call structure. There is one restriction we call the 3d restriction which means a containment or stacking of nodes. The o restriction is a combination
of x and y. The elements are arranged in a circle around a center point. An example of a constellation diagram is shown in Figure 4.4.

### 4.2.1 Semantically unrestricted diagrams

![UML Class Diagram](image)

To sum it up, there are diagram types that are restricted in their layout and there are diagrams that have no such restrictions. In the following we present examples of the two coarse-grained categories and how they affect our algorithms and layout, respectively. Firstly, we refer to the diagrams that can be arranged freely on the canvas and that are decoupled from the semantics of the underlying model they are a representation of. These semantically unrestricted diagrams however, can be restricted with regard to their layout as well which is why we subdivided this category further.
into graphically unrestricted, partially restricted, and highly restricted diagrams.

**Figure 4.6: UML Activity Diagram**

**Graphically Unrestricted Diagrams.** Diagrams that have no restrictions at all leave us the option to apply layout adjustment algorithms of our choice. One example for an unrestricted diagram is a UML class diagram like the one depicted in Figure 4.5. Including packages in this diagram however—as is possible in a class diagram—would add a layout restriction. Therefore, these do not belong in this category.

**Figure 4.7: UML Sequence Diagram**

A second example for unrestricted diagrams is an activity diagram like the one depicted in Figure 4.6. Note that this category is not concerned with the edges of diagrams. This means that edges may overlap although that can reduce the readability.

**Partially Restricted Diagrams.** There is at least one constraint that has to be kept in partially restricted diagrams. For example, such a constraint is the 3d alignment in CIM diagrams (cf. Figure 4.2) or class diagrams with packages. The root nodes can be aligned freely as well as the child nodes but the important thing is that the children are within the bounds of the parent.
4.2. DIAGRAM CATEGORIES

Highly Restricted Diagrams. There are highly restricted diagrams like the sequence diagram in Figure 4.7. These hardly allow for any rearrangement of the layout. In sequence diagrams, the objects have a restriction regarding the y value \((y = 0)\) and they are ordered in x direction as well to avoid edge crossings. These edges symbolize messages in this kind of diagram and it would be hard, if not impossible, to understand it without these restrictions. Additionally, these messages have a certain order on the object’s life line (the y direction).

As shown in Table 4.1 there are two kinds of use case diagrams. One is more restricted than the other. Figure 4.8 depicts the highly restricted kind of such a diagram. The actors and the use case box in this example are ordered horizontally and the use case ellipses are within the system rectangle. This nesting restriction is the one we call 3d.

4.2.2 Semantically restricted diagrams

In the previous section, we presented some examples of semantically not restricted diagrams. This decoupling of semantics and layout is achieved by using a separate layout language as depicted in Figure 4.1. In this section however, we discuss representations of a model which are semantically coupled with the model. These do not have an extra layer for the representation.

Our work is based on Ecore models with EMFText [HJK+09] syntax. Using the EMFText-based JaMoPP [GDc] to handle Java code as model, for example, does not change the semantic restrictions of Java.
CHAPTER 4. MODELING

### Table 4.2: categories of diagram layout restrictions

<table>
<thead>
<tr>
<th>semantic. unrestricted</th>
<th>unrestricted</th>
<th>partially restricted</th>
<th>highly restricted</th>
<th>semantic. restricted</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class Diag.</td>
<td>Class Diag./Packages</td>
<td>Sequence Diag.</td>
<td>Text</td>
<td></td>
</tr>
<tr>
<td>Activity Diag.</td>
<td>CIM Diag.</td>
<td>Use Case Diag.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Use Case Diag. (rest.)</td>
<td>Constellation Diag.</td>
<td>Activity Diag/SL.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Deployment Diag.</td>
<td>Family Tree</td>
<td>Ishikawa Diag</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Euler Diag</td>
<td>Gantt Chart</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Listing 4.1: sample Java code

```java
int act = 0;

if (fragit.hasNext()) {
    FragmentDiagram diag = fragit.next();
    while(fragit.hasNext()){
        act++;
        FragmentDiagram diag2 = fragit.next();
        // setting the same space between two fragments
        // in the result that was set in the target info
        int oldx = diag2.getTargetBounds().getX();
        if (oldx <= 5) {
            oldx = oldx * 5;
        }
        diag2.getTargetBounds().setX((oldx+(act*overlap)));
    }
}
```

Listing 4.1 shows a snippet of Java code from our implementation. If this was the outcome of a model transformation and we had no restrictions regarding the layout we could have introduced the variables `act`, `diag`, and `diag2` after Line 16. However, this is not possible in Java and causes an error since the variables have to be introduced before they are used.

Finally, in Table 4.2 we provide an order of the different diagram types regarding their layout restrictions. We sorted them into the categories we found but we do not claim it to be exhaustive. However, in this work we concentrate on finding strategies for the layout adjustment of the different categories introduced in this section. Having four categories requires a rather flexible approach to layout management. In Chapter 5 we present the modifications we applied to our layout adjustment framework from [Gau09] with respect to these categories and in Chapter 6 we discuss an evaluation we did with probands in order to find out how good the diagram types are supported by our adapted solution.
Chapter 5

Architecture and Implementation

Until now we introduced the foundations for our work and the notion of diagram categories that we need to implement in order to create a framework which is user friendly and widely usable.

As mentioned, the framework we developed for graphical layout languages had to be adapted. In the process of our work we constantly discovered new use cases that had to be covered by our implementation. Therefore, we required new components and even a new approach. One important fact is that we have to deal with textual as well as graphical languages now. Firstly, as discussed in Chapter 2, we had to implement providers for all the languages we require layout information of during our transformation process. Since we already have those for the languages with graphical syntax (GMF and TOPCASED) we had to add some for textual languages. Having a EMFText syntax, both Java and UseCaseInvariant require the same providers although the target provider is never used for Java in our example. EMFTextTip and EMFTextSip are discussed in further detail in Section 5.2.3.

In this chapter we present our approaches and solutions as well as the architecture of the prototype we implemented. In order to adapt the framework we had to invent a new architecture of Providers, Arrangers and Mergers. This adaptation is discussed in this chapter. Further, we give a more detailed description of the single components that have been implemented.

This chapter is structured as follows. In Section 5.1 we introduce LaCoMe (Layout Composition Framework), the extended layout composition framework. Since it has been decoupled from Reuseware we had to come up with this new name. Afterward, in Section 5.2 we present the components of LaCoMe and how they work.

5.1 LaCoMe - Extending the Extension

This first section discusses the advancement of the layout composition framework with respect to the architecture. In [Gau09] we started with a composition process
In which users were given a graphical editor to lay out illustrations of fragments they wanted to compose. The graphical syntax of the layout languages allowed for easy layout information gathering and, of course, adjustment. Furthermore, the diagrams we adjusted were hardly constrained which did not require any consideration of special adjustment strategies. In this current work, however, we faced more difficult tasks and thus needed to extend the layout composition framework.

Figure 5.1: LaCoMe structure

Figure 5.1 shows the structure of LaCoMe. We implemented three parts; the basic components, i.e. the framework itself, which can be found in the base directory, the layout adjustment components in `lacome.adjustment.*`, and the layout language specific classes in `lacome.layoutlanguage.*`. The basis of LaCoMe is built of some interfaces for Comparator, Arranger, Merger, and Provider as well as the classes `FragmentDiagram`, `Bounds`, `CompositionDiagram`, and `DiagramCompositionStrategy`. The latter is a new component which replaces the `DiagramComposer` of [Gau09]. Why this was necessary is discussed in this chapter. Furthermore, a new component named `TraceProvider` was implemented which was required to decouple LaCoMe from Reuseware.

At first, Section 5.1.1 discusses the adaptation of `FragmentDiagram` and `Bounds`, components we introduced in [Gau09] as well and which are crucial if we want to understand LaCoMe. Secondly, in Section 5.1.2 the changes applied to `CompositionDiagramUtil` are presented.

### 5.1.1 Extension of FragmentDiagram and Bounds class

To start with the presentation of the changes we applied to the layout composition framework, we first present the classes `FragmentDiagram` and `Bounds`. We had to adjust these to meet the new requirements of our extended and improved approach.

The `FragmentDiagram`, one of the basic components of our framework, changed compared to the one we presented in [Gau09].
5.1. LACOME - EXTENDING THE EXTENSION

```java
/**
 * Information about a fragment diagram.
 */
public class FragmentDiagram {

    public static final String DEFAULT_STRATEGY_ID = "default";

    protected EList<EObject> contents;
    protected EList<EObject> diagramRoots;
    protected EObject fragmentInstance;
    protected Bounds sourceBounds;
    protected Bounds targetBounds;
    protected String strategy;

    public FragmentDiagram(
        EList<EObject> contents,
        EList<EObject> diagramRoots,
        EObject fragmentInstance,
        String strategy);

    /**
     * @return The actual fragment model
     */
    public EList<EObject> getContents() {
        return contents;
    }

    /**
     * @return The actual diagram information for the fragment
     */
    public EList<EObject> getDiagramRoots() {
        return diagramRoots;
    }

    /**
     * @return fragment instance that provides the target information
     */
    public EObject getFragmentInstance() {
        return fragmentInstance;
    }

    /**
     * @return The the source diagrams bounding box as computed from
     * the diagram
     */
    public Bounds getSourceBounds() {
        return sourceBounds;
    }
}
```
As is shown in Listing 5.1, we have one constructor left where we had three in the previous version. Although that seems like a reduction it is quite the opposite. All that is required is now provided by this single constructor. The content is the model, the diagramRoots are required for the diagram and the fragmentInstance is the link to the context of a FragmentDiagram. Of course the getter for source and target bounds are still required as well. An attribute we did not have before is the strategy. We learned that it is inevitable to apply the Layout Composition Strategy to each FragmentDiagram instead of just arranging the whole composition program. Having different sources for a composition often required the employment of different strategies for each of the components in order to satisfy the layout restrictions imposed by the diagram categories.

Additionally, we had to adapt the Bounds class to our new requirements. In a specific example, the diagram was not laid out using pixels for the measurement unit but millimeters. Thus, we had to add getters and setters for the measurementUnit in order to recalculate the bounds if a different measurement unit than the default is detected.
Listing 5.2: recalculations due to measurement unit in GMFMerger

Listing 5.2 shows the recalculation of x and y values when the measurement units of source and target diagram differ. In Lines 2 and 3 source and target measurement unit are checked for the default MU_NULL. If these two measurement units differ and the source fragment is in pixels, x and y are recalculated in Lines 6-9. LPtoDP() is a method from draw2d (GMF) which transforms logical units, i.e. pixels, to device units, i.e. millimeters. If the source fragment is in himetric, x and y are recalculated according to DPtpLP(). We require a default value because textual languages are not displayed using measurement units. Setting one in the information providers just to make it fit was not an option.

On the other hand, the modelPosition attribute was introduced. As mentioned, there are different types of use cases for our framework. We can gather graphical information from diagrams and rearrange nodes setting their x and y values. These graphical diagrams however, are abstractions of the real model (cf. Figure 4.1). Not having such an abstraction—as it is the case for textual modeling languages—requires a different approach. In Reuseware, model elements have an order inside models and hence we can get the position of the element in that list for our layout adjustment. Another option will be discussed later.

5.1.2 Extension of CompositionDiagramUtil

The previous section covers the refinements applied to FragmentDiagram and Bounds, two classes required for layout composition strategies, the rearrangement of diagrams and the gathering of layout information. Now we present the CompositionDiagramUtil class which is one of the central components of our framework.

In [Gau09] we presented the class CompositionDiagramUtil and we stated that this class is required for orchestrating the whole composition process. The execution order of the single actions in that process is still of great importance if we want to get the results we are aiming at with our work. Yet there had to be several adaptations due to the new layout composition strategies.

```java
for (String arrangerID : diagramCompositionStrategy.getArrangerIDs()) {
    DiagramArranger<? super MultiSourceDiagramArranger> arranger = diagramArrangers.get(arrangerID);
    if (arranger instanceof MultiSourceDiagramArranger) {
        if (!contributingDiagramsWithStrategy.isEmpty()) {
            // sort the list of contributing diagrams
            Collections.sort(contributingDiagramsWithStrategy, comparator);
            ((MultiSourceDiagramArranger) arranger).arrange(
```
CHAPTER 5. ARCHITECTURE AND IMPLEMENTATION

Listing 5.3 shows the call of arrange for MultiSourceDiagramArrangers with its new preconditions. In our previous work we had to check whether the assigned Arranger was single or multiple and handle the differences. Now, however, with a composition strategy which can consist of more than just one arranger, there are more preconditions. We have to get all arrangerIDs that have been assigned to a composition strategy and create Arrangers for each of these IDs (Lines 1-2). While in [Gau09] a Comparator was an attribute of diagramComposer (cf. Section 2.2.6) we now decided to make it an attribute of the Arranger. We could also have assigned it to the whole strategy but this current version makes more sense since each of the Arrangers may require a different Comparator. Although applying an Arranger without a Comparator might yield unwanted results we decided to make that attribute optional. In our previous work it was a required attribute because there were no cases for which it made sense to omit this step of sorting the composition steps. This was provoked by the indeterministic way Reuseware has of choosing the next fragment for a composition. For graphical layout languages this indeterminism caused different results each time the layout adjustment was executed. Having textual languages and the modelPosition attribute, however, sorting the composition steps is not always required. For SingleSourceDiagramArrangers the conditions changed equivalently.

In order to initialize the extension points we have to call the method initEPs() (not shown). This has to be done prior to the call of composeDiagrams. Skipping this would cause problems since none of the components we mean to use would be known to Reuseware.

```
private static void initDiagramCompositionStrategies() {
    if (Platform.isRunning()) {
        // read extension point
        IExtensionPoint diagramCompositionStrategyEP =
            Platform.getExtensionRegistry().
                getExtensionPoint(DIAGRAM_COMPOSITION_STRATEGY_EP_ID);
        IConfigurationElement[] entries = diagramCompositionStrategyEP
            .getConfigurationElements();

        for (int i = 0; i < entries.length; i++) {
            try {
                String ID = entries[i].getAttribute("id");
                DiagramCompositionStrategy compositionStrategy =
                    new DiagramCompositionStrategy(ID);
```
5.2 Implemented Components

In this section we present the new components compared to [Gau09] applied to the framework which provide support even for textual languages. Furthermore, the adaptations required for the composition of diagrams from different categories (c.f. Section 4.2) are discussed.

Firstly, we tried whether it is possible to adjust the framework at hand in a way to work with textual languages. Therefore, the TargetDiagramInformationProvider and the SourceDiagramInformationProvider had to be adapted. In Section 3.2, we presented the motivating example which forms the basis for our work and requires the new components we introduce in the following.

```java
for ( IConfigurationElement arranger : entries[i] 
    .getChildren("arranger") ) { 
    if (arranger.getAttribute("id") != null) { 
        String arrangerID = arranger.getAttribute("id"); 
        compositionStrategy.getArrangerIDs().add(arrangerID); 
    }
}

for ( IConfigurationElement deactivateMerger : entries[i] 
    .getChildren("deactivateMerger") ) { 
    if (deactivateMerger.getAttribute("id") != null) { 
        String mergerID = deactivateMerger.getAttribute("id"); 
        compositionStrategy.getDeactivateMergerIDs().add(mergerID); 
    }
}

    diagramCompositionStrategies.put(ID, compositionStrategy);
}

Listing 5.4: initDiagramCompositionStrategy() of CompositionDiagramUtil

Listing 5.4 shows the initialization of the new extension point DiagramCompositionStrategy. This method is called in initEPs() and accesses the extension point via its ID DIAGRAM_COMPOSITION_STRATEGY_EP_ID (Lines 4-6). An extension point, as mentioned in 2.2.6, conforms to a XML schema definition (EXSD) [Con] and we can traverse the entries of an extension point in order to create objects. In Lines 13 to 15 we create a DiagramCompositionStrategy with the id attribute we get from the current entry. Since Arrangers are children of strategies in the schema we can get them with the for statement in Lines 17 and 18. As stated above, there can be many Arrangers in a layout strategy. Therefore, we add all IDs of the assigned Arrangers to the arrangerIDs list. If this list is empty, i.e. no arrangers were specified in a strategy, the diagram arrangement shown in Listing 5.3 is not executed.
CHAPTER 5. ARCHITECTURE AND IMPLEMENTATION

5.2.1 TraceProvider

Having decoupled LaCoMe from Reuseware required the implementation of additional components such as the TraceProvider. Some of LaCoMe’s classes use Reuseware specific methods in order to, for example, get access to the original of a fragment. As mentioned, fragments in a composition process are copied and composed which enables us to reuse one fragment many times. However, a connector to Reuseware was required. This connection is provided by the TraceProvider.

```java
package org.reuseware.lacome;
import java.util.List;
import org.eclipse.emf.ecore.EObject;
import org.eclipse.emf.ecore.EStructuralFeature.Setting;

public interface TraceProvider {
    List<EObject> getReplacedValues(EObject element);
    EObject getOriginal(EObject copy);
    Setting getRemovedFromSetting(EObject element);
    List<EObject> getDerivedFrom(EObject context);
}
```

Listing 5.5: TraceProvider interface

In Listing 5.5 this TraceProvider interface is shown. The methods of this component are discussed later. On the other side, i.e. within Reuseware, there is the class CoCoNutTraceProvider which implements our interface and provides the required access to Reuseware components and methods.

5.2.2 DiagramCompositionStrategy

As stated, we replaced the DiagramComposer with a new class. Besides the new name—DiagramCompositionStrategy—which strikes closer to the actual tasks it has to perform, some other differences can be pointed out.

```java
public class DiagramCompositionStrategy {
    protected String ID;
    protected List<String> arrangerIDs = new ArrayList<String>();
    protected List<String> deactivateMergerIDs = new ArrayList<String>();

    public DiagramCompositionStrategy(String ID) {
        super();
    }
}
```
5.2. IMPLEMENTED COMPONENTS

As is shown in Listing 5.6, this class provides a list each of Arrangers and of DeactivateMergers. What an Arranger does should be quite clear but Deactivate-Merger has not been mentioned yet. Basically, these IDs are of Mergers that are specified in a new extension point called diagramCompositionStrategy. Within that extension point, users can list a number of Arrangers to be applied and a number of Mergers not to be applied when this strategy is chosen for the layout adjustment. This notion of DeactivateMergers is further discussed in 5.2.8.

5.2.3 Information Provider

Since six languages are employed in the motivating example of Section 3.2 whereof some have different layout languages, information providers for each layout language are required. However, for both GMF and TOPCASED we implemented these providers in [Gau09]. Therefore, the following classes have to be added in order to cover all layout languages.

- ODTTargetInformationProvider (ODTTip)
- EMFTextTargetInformationProvider (EMFTextTip)
- EMFTextSourceInformationProvider (EMFTextSip)

There is no need to add a source information provider for OpenOffice documents in the example but it can be added when required.

As discussed in Section 3.2, OpenOffice documents serve for use case descriptions. These documents conform to the OASIS OpenDocument RelaxNG schema which was translated into an XSD Schema using Trang. Converting this with the XSD-to-Ecore binding of EMF results in an Ecore metamodel.

Within the OpenOffice document, special styles can be created the actors of the use cases can be marked with. In our case, this style is named Actor. Once the text is marked that way, we are able to search for it by iterating over the document’s
Figure 5.2: Structure of content.xml

structure. Figure 5.2 depicts the content.xml\(^1\) of the BookTicket.odt. We searched the whole file top down for all so called SpanTypes.

```
private int findOdt(EOBJECT context) {
    if (!CompositionDiagramUtil.getDerivedFrom(context).isEmpty()) {
        EOBJECT semanticElement = CompositionDiagramUtil.getDerivedFrom(context).get(0);
        if (!((semanticElement instanceof SpanType)) {
            return -1;
        }
        EOJECT container = EcoreUtil.getRootContainer(semanticElement);
        ELIST<SPAN_TYPE> spanList = new BasicEList<SPAN_TYPE>();
        for (ITERATOR<EOBJECT> i = container.eAllContents(); i.hasNext(); ) {
            EOJECT nextEl = i.next();
            if (nextEl instanceof SPAN_TYPE) {
                SPAN_TYPE span = (SPAN_TYPE) nextEl;
                spanList.add(span);
            }
        }
    }
}
```

\(^1\)An ODT file is a zip archive with xml documents of which content.xml contains the actual content of the document.
5.2. IMPLEMENTED COMPONENTS

In Listing 5.7 we can see how this search is done. The expression `CompositionDiagramUtil.getDerivedFrom(context).get(0)` returns the first element of the derived from reference list. Now this is one of the methods presented in Section 5.2.1. Figure 5.3 depicts part of the composition system from Section 3.1. `getDerivedFrom` is a method within Reuseware which provides a link from the composition program to the composition language. The context is checked whether it is of the type `FragmentInstance` (cf. Section 2.1.1) and returns the element this `FragmentInstance` was derived from.

![Figure 5.3: The getDerivedFrom relation in Reuseware](image)

From this `semanticElement` we can get the root of the document by calling `EcoreUtil.getRootContainer(semanticElement)`. In the `for` statement we iterate over the whole content of that list in order to find items of type `SpanType` which we add to an `EList`. Since we start at the root of the model these items are added to the list in the same order they have in the OpenOffice document. After the whole document is searched and the list is complete, we return the index of the `semanticElement` which will then be one value of the target bounds of the current `fragmentDiagram`. Since we do not extract any more values from the odt file so far, this model position is `x`, `y` and `modelPosition` value as well. However, only the `y` value is used in our example.

Looking at our example from Section 3.2, from `BookTicket.odt` we get the information for the actors `Customer`, `Clerk`, and `Bank`. However, the example has yet another actor called `Hall`. Since this actor is not specified in the odt file (cf. Figures 3.7(b) and 3.8) because it is a different concern, we have to gather the required information from another source. The additional information provider has to be able to deal with an EMFText syntax since the `UseCaseInvariant` language has one such.
In 4.2 we mentioned that x values in a graphical syntax are equivalent to the columns in a textual syntax and the y values are equivalent to lines. However, we still use the `modelPosition` for the text because we discovered that it better fits the purpose of a provider. This component should not care about how the provided information is used in the end but simply get all information there is. Furthermore, this `modelPosition` is required for a textual syntax because no arbitrary rearrangement is allowed here. We discussed that issue in Section 4.1. Due to the lack of an abstraction layer like in graphical modeling the text is semantically restricted and the rearrangement would influence the model.
5.2. IMPLEMENTED COMPONENTS

Listing 5.8 presents the gathering of that model position for EMFText sources. In Line 10 the operation `getFirstSource` is called. After the context is checked for null in Lines 18-21 we return the original source if it exists and the copy if there is no original. This check is important because the fragments not always are the original in a Reuseware software development process. `getOriginal(copy)` is one of TraceProviders’ methods which link LaCoMe to Reuseware. Within CoCoNut there is a class `SyncEcoreUtil` where `getOriginal` is declared. If there is an original for the `FragmentInstance`, this is used. Otherwise, the copy is the basis for the information gathering.

Since we now have the target information for this step in our MDSD process we also require the source information for the layout composition. Being an example with many concerns, the ticket shops’ layout information depends on the representation we want to get. Whatever fragments we enrich the composition program (cf. Section 3.2) with provide the layout information for our target model and thus, we have to write SIPs for these languages. In [Gau09] we developed SIPs for GMF [Foub] and TOPCASED [Val] which we can use without having to adapt anything. With these providers, all graphical layout languages of our example are covered. This leaves us the textual languages, Java and UseCaseInvariant. Since both have a EMFText syntax, a SIP for this language had to be developed.

```java
public void provideBounds(FragmentDiagram fragmentDiagram) {
    EObject modelElement = fragmentDiagram.getDocumentRoots().get(0);
    Resource resource = CompositionDiagramUtil.getOriginal(modelElement).eResource();
    IResource textResource = castToTextResource(resource);
    int line = textResource.getLocationMap().getLine(
            CompositionDiagramUtil.getOriginal(modelElement));
    fragmentDiagram.getSourceBounds().setModelPosition(line);
}
```

Listing 5.9: Method provideBounds(FragmentDiagram) of EMFTextSip

Listing 5.9 provides the necessary code for getting access to the model information in EMFText. The `modelElement` is a EMFText model element we want to have information about. By the Lines 4-5 we get access to the original text resource of that particular model element. The information we retrieve from that text resource is the line number of the model element which we assign to the fragment diagram’s `modelPosition` value in Line 11.

The operation call in Line 6—the `castToTextResource`—is responsible for casting the `EObject` `Resource` to an EMFText `IResource`. Therefore, we check whether the class of the Ecore `Resource` has a method called `getLocationMap` in Line 4 of Listing 5.10. When this condition is satisfied, we return the `IResource` of it.
private IResource castToTextResource(Resource resource) {
    try {
        // check if this resource provides "getLocationMap"
        resource.getClass().getMethod("getLocationMap", new Class[]{}) ;
    } catch (Exception e) {
        return null;
    }
    return (IResource) EMFTextAccessProxy.get(resource, IResource.class);
}

Listing 5.10: Method castToTextResource(Resource) of EMFTextSip

Depending on which concerns are activated, the source information changes. However, with the providers for GMF, TOPCASED, ODT, and EMFText, we cover all languages employed in our example. Once new languages are introduced in other examples, new providers have to be added for those.

5.2.4 Merger

![Diagram](image)

Figure 5.4: getRemovedFromSetting()

As discussed in Section 2.2, one more component of our layout adjustment extension has to be adapted to the new language. This Merger is responsible for materializing the changes the Arranger made. There is a Merger required for every layout language and for the graphical languages we developed those in [Gau09]. Thus, one Merger for languages with EMFText syntax had to be added.

private void mergeEMFTextEObject tRootElement, int modelPosition) {
    TreeIterator<EOBJECT> iterate = EcoreUtil.getAllContents(Collections.singletonList(sRootElement));
    while (iterate.hasNext()) {
        EObject sourceNext = iterate.next();
        TreeIterator<EOBJECT> targetIt = tRootElement.eAllContents();
        while (targetIt.hasNext()) {
            EObject tnext = targetIt.next();
        }
    }
}
5.2. IMPLEMENTED COMPONENTS

```java
Setting sourceSetting = CompositionDiagramUtil
    .getRemovedFromSetting(tnext);
if (sourceSetting != null) {
    if (sourceNext.equals(sourceSetting.getEObject())) {
        EReference tnextRef = tnext.eContainmentFeature();
        if (tnextRef.isMany()) {
            @SuppressWarnings("unchecked")
            EList<EObject> containerlist = (EList<EObject>) tnext
                .eContainer().eGet(tnextRef);
            containerlist.move(modelPosition, tnext);
            iterate.prune();
            break;
        }
    }
}
```

Listing 5.11: Method mergeEMFText(EObject, EObject, int) of EMFTextMerger

We require the root element of both the source fragments and the target model. These are input to the method `mergeEMFText()` shown in Listing 5.11. The third value `modelPosition` is the one we assigned to the `fragmentDiagram` using the ODT-TargetInformationProvider. This value should now determine the order of the new document's elements. In `mergeEMFText()` we iterate over the original source fragment's content as well as over the target model's content and once two elements are equal (cf. Line 12) the element of the composed model is moved to the position given by `modelPosition`.

At least this was the approach we used before we refactored the framework. Actually, what this Merger does in that method usually is the task of an Arranger. There is no need to merge any diagram because there is no real diagram for textual models. These models just have to be arranged and after that, the layout adjustment is done. How we overcame this will be discussed in Section 5.2.8. In the next Section we present the `ModelArranger`.

5.2.5 ModelArranger

One reason for the refactoring of the layout composition framework was the `EMFTextMerger`. When we were done implementing it, we discovered that it did a task it was never designed for. When we wanted a text model rearranged, the Arranger had to be disabled—or none had to be specified—in the composition strategy but the Merger had to be employed. This Merger was a mandatory attribute in our previous work anyway. Since even Arrangers could be left out, we implemented it this way. However, we realized that the models of other fragments may require a rearrangement too, although it is not the case in our example. This use case made us reconsider the approach because it would be against our definitions to use the `EMFTextMerger` when a model requires rearranging which has a GMF syntax for example. Therefore, we implemented the `ModelArranger`.
We copied the code from Listing 5.11 and inserted it into a new class. In contrast to Mergers there is no `canArrange()` check in Arrangers since they are layout language independent. Therefore, this arranger can be applied to model fragments no matter which layout language they have.

```java
public static Setting getRemovedFromSetting(EObject replacement) {
    if (replacement == null) {
        return null;
    }
    for (Adapter a : replacement.eAdapters()) {
        if (a instanceof ReplacementForAdapter) {
            return ((ReplacementForAdapter)a).getRemovedFromSetting();
        }
    }
    return null;
}
```

Listing 5.12: Method `getRemovedFromSetting(EObject)`

In Section 5.2.4 we explained how the merge method (cf. Listing 5.11) works and since it is copied the Arranger does exactly the same as the Merger. In Listing 5.12 the method `getRemovedFromSetting()` is shown which we use in the Arranger in Line 11. This method returns the `Setting` which contains the model element that has been removed from the source in order to add it to the target. Figure 5.4 depicts the methods’ purpose. In Figure 5.4(a) both the source and the target setting are shown before a composition. Figure 5.4(b), on the other hand, depicts the result of a composition. `tnext` is an element in the composed model which we require the `sourceSetting` of in order to compare it to the current source element `sourceNext`. By this check we want to make sure we assign the `modelPosition` of the current source element to its equivalent in the target model.

### 5.2.6 MultiHorizontalArranger

![Figure 5.5: (a) Overlap with bounding box (b) adjusted diagram](image)

In [Gau09] we presented two different kinds of Arrangers, `SingleSourceDiagramArranger` and `MultiSourceDiagramArranger`. The naming scheme indicates their usage. While `SingleSourceDiagramArrangers` see only one fragment diagram at a
5.2. IMPLEMENTED COMPONENTS

time, MultiSourceDiagramArrangers have a list of all fragment diagrams involved in the composition. Having to treat only one diagram at a time might speed up the whole process of finding overlaps. However, since we merge after every arrangement step (cf. Sections 2.2.3 and 5.1.2) we have no notion of what the outcome of the whole composition might be until it is finished. Furthermore, in our HorizontalArranger (cf. Section 2.2.7) which was a SingleSourceDiagramArranger we had to check for overlaps with the bounding box of the target diagram (cf. Figure 5.5). This bounding box got bigger and bigger each time we added a fragment diagram. Hence, even though the next fragment diagram does not overlap with any previous elements, it did overlap with the bounding box and had to be rearranged. Figure 5.5(b) depicts the rearrangement of fragment F3 with the HorizontalArranger.

This flaw can be overcome by the MultiSourceDiagramArrangers. There is no merge after each arrangement and we do not check for overlaps with the bounding box of the target diagram. Here we check for overlaps between the real elements of a diagram. We learned that this approach yields better results

```java
private int isOverlap(List<FragmentDiagram> diagram)
{
    ListIterator<FragmentDiagram> fragit = diagram.listIterator();
    int overlap = 0;
    int actDiag = 0;

    while (fragit.hasNext()) {
        actDiag++;
        FragmentDiagram source1 = fragit.next();
        for (int i = actDiag; i < diagram.size(); i++){
            FragmentDiagram source2 = diagram.get(i);

            // definition of the center points of the involved nodes
            double source1x = (source1.getTargetBounds().getX() + (source1.getSourceBounds().getWidth() / 2));
            double source1y = (source1.getTargetBounds().getY() + (source1.getSourceBounds().getHeight() / 2));
            double source2x = (source2.getTargetBounds().getX() + (source2.getSourceBounds().getWidth() / 2));
            double source2y = (source2.getTargetBounds().getY() + (source2.getSourceBounds().getHeight() / 2));

            double absolutex = Math.abs(source1x - source2x);
            double width = (source1.getSourceBounds().getWidth() + source2.getSourceBounds().getWidth()) / 2;
            double absolutey = Math.abs(source1y - source2y);
            double height = (source1.getSourceBounds().getHeight() + source2.getSourceBounds().getHeight()) / 2;

            int over = (source1.getTargetBounds().getX() + source1.getSourceBounds().getWidth()) - source2.getTargetBounds().getX();

            // two fragment diagrams do not overlap if
            // |x1−x2|>= (w1+w2)/2 or |y1−y2|>= (h1+h2)/2
```
Listing 5.13: Method isOverlap(List<FragmentDiagram>) of MultiHorizontalArranger

Listing 5.13 shows the overlap check for MultiSourceDiagramArrangers. This is an improved check compared to the one we used in [Gau09]. As of now it is not only possible to check whether two fragments overlap or not but even how much they overlap. For the UniformScalingArranger—the only MultiSourceDiagramArranger at that time—in [Gau09] this was not necessary since the adjustment was independent of any of these overlap spaces. There we specified a scaling factor and the adjustment was just triggered by the isOverlap method. MultiHorizontalArranger however, requires data about the overlap in order to adjust it correctly.

According to [MSTH03, HL03], two nodes do not overlap if the following constraints are satisfied

\[ |x_v - x_u| \geq \frac{(w_v + w_u)}{2} \lor |y_v - y_u| \geq \frac{(h_v + h_u)}{2} \]

where \( u, v \in V \) and \( V = \{1, ..., n\} \) is a set of nodes in a graph \( G = (V, E) \). In isOverlap we always compare the current fragment diagram with other elements in the given list. Prior to this check, the list was sorted by a Comparator which is why the fragment diagram is not compared to all but only to its successors in the list. Line 8 says that the current fragment diagram is named \( source1 \) and in the Lines 9 and 10 we iterate over the successors with a for-statement. The next step was to gather the center points of both elements which is done in the Lines 13-20. Afterward, in the Lines 22-27, we calculate the values for \( |x_v - x_u|, \frac{(w_v + w_u)}{2}, |y_v - y_u| \) and \( \frac{(h_v + h_u)}{2} \). Before checking the condition in Line 35 we define the actual value of the overlap.

Figure 5.6 depicts two overlapping fragment diagrams and the required information for eliminating this overlap. In the MultiHorizontalArranger we want to know whether two fragment diagrams overlap and if they do, how much. Since we want to arrange horizontally, the width is the important value for the adjustment but the overlap check requires both width and height (not shown in Figure 5.6).

5.2.7 MultiVerticalArranger

For some diagrams a vertical sorting serves better than a horizontal one. This is why we implemented the MultiVerticalArranger and the VerticalArranger. The first is a MultiSourceDiagramArranger while the latter is a SingleSourceDiagramArranger.
Both these Arrangers require a new composition order. We discussed that issue in the conclusion of [Gau09]. Using the XYPositionComparator of our previous work would sort the elements in x direction. However, this would yield unwanted composition results if applied for a vertical arranger. Therefore, we had to implement a YPositionComparator which compares the elements in y direction.

```java
private void sortVertically(List<FragmentDiagram> diagram, int overlap) {
    ListIterator<FragmentDiagram> fragit = diagram.listIterator();
    // the factor for the overlap. if one fragment
    // has been moved, the next has to be moved by that amount
    // plus its own overlap
    int act = 0;

    if (fragit.hasNext()) {
        FragmentDiagram diag = fragit.next();
        while (fragit.hasNext()) {
            act++;
            FragmentDiagram diag2 = fragit.next();
            // setting the same space between two fragments
            // in the result that was set in the target info
            int oldy = diag2.getTargetBounds().getY();
            if (oldy <= 5) {
                oldy += 5;
            }
            diag2.getTargetBounds().setY((oldy + (act * overlap)));
        }
    }
}
```

Listing 5.14: Method sortVertically(List<FragmentDiagram>)

When the composition order is correct, the MultiVerticalArranger can be applied. Firstly, an overlap check like the one in Listing 5.13 has to be done in order to calculate the overlap space of two fragments. For this, the overlap calculation
in Lines 19-21 of Listing 5.13 has to be adapted. The second change is applied to the if statement in Line 26. In both, calculation and if statement, width has to be replaced by height and x has to be replaced by y. Afterward, when the overlap amount is returned, the method sortVertically resets the y value for the elements of the fragment diagrams. Of course, sortHorizontally resets the x value in the MultiHorizontalArranger.

![Diagram](image)

Figure 5.7: (a) Diagram before adjustment (b) incomplete adjustment (c) complete adjustment

Listing 5.14 shows the method sortVertically. In Line 3 we define an iterator for the list of FragmentDiagrams which are involved in the process. In Line 7 a variable act is defined which is required for rearranging all successors of the current diagram by the same amount. The value of this variable is increased by 1 for each cycling of the while statement in Line 11. The reason for this is depicted in Figure 5.7. If we only adjusted fragment diagram F2 by the amount it overlaps with F1 and skip the adjustment of F3 the overlap between F1 and F3 would not be detected and hence, the composed diagrams’ layout would not be adjusted properly. This is shown in Figure 5.7(b). Since F3 is the third fragment diagram in the composition order however, taking the overlap value F2 is enough to eliminate a possible overlap of F3 as well. Furthermore, this keeps the neighborhood relationship of all involved fragment diagrams (cf. Figure 5.7(c)).

In our example, fragments are but small and often the source information and target information is the same for the x and y value. The texts we retrieve our information from are at most five lines long and we have a maximum of three actors in a list of SpanTypes. For layout purposes this is not much and since we want to be able to distinguish the different elements we had to multiply the given information (Lines 17-19). This adds some space between two elements.
5.2. IMPLEMENTED COMPONENTS

5.2.8 LayoutCompositionStrategy Extension Point

In Section 2.2.6 we introduced the DiagramComposer extension point we implemented in [Gau09]. As stated, this extension point enabled users to choose the components they want to employ for the layout adjustment process. This DiagramComposer had four attributes: id, comparator, arranger, and merger. If we wanted to have a different arrangement, we had to add a new DiagramComposer.

In our current work however, we discovered that this extension point is not enough to cover the diagram categories presented in Section 4.2. We learned that it had to be possible to employ more than just one Arranger and that a Merger is not always necessary. Furthermore, the Comparator is a component that should not be composer specific but Arranger specific.

One exemplary strategy is shown in Listing 5.15. Since it was named a Layout Composition Strategy we decided strategy was more intuitive than composer and renamed it. However, the attributes of strategy are quite familiar although we removed arranger and merger. The latter was removed because we found that merging can be done similar to the diagram information provision. If the canMerge condition is satisfied, the Merger is executed. For some cases however, a Merger must not be executed which is why we inserted a new element with * multiplicity, the deactivateMerger element. In CompositionDiagramUtil we collect all Mergers which satisfy canMerge but only execute the ones not in the deactivateMerger list.

```
<strategy
  id="mhsort">
  <arranger
    id="id"
    comparator="org.example.ExampleComparator">
  </arranger>
  <deactivateMerger
    id="id">
  </deactivateMerger>
</strategy>
```

Listing 5.15: DiagramCompositionStrategy

```
for (String mergerID : diagramMergers.keySet()) {
  if (!diagramCompositionStrategy.getDeactivateMergerIDs().
    contains(mergerID)) {
    DiagramMerger merger = diagramMergers.get(mergerID);
    if (merger.canMerge(receivingDiagram)) {
      merger.merge(contributingDiagramsWithStrategy,
        receivingDiagram,
        new BasicEList<FragmentDiagram>(()));
    }
  }
}
```

Listing 5.16: check for deactivateMerger in CompositionDiagramUtil
Listing 5.16 shows the check for Mergers which should not be executed. `getDeactivateMergerIDs()` in Line 2 returns a `List<String>` with all `deactivateMerger` elements of the `DiagramCompositionStrategy`. If the current `mergerID` is not in that list, the merger with this id is created in Line 4.

Additionally, we defined an extension point for Arrangers which also is a new element of the composition strategy. Having this extension point allows for the combination of Arrangers. Thus, its multiplicity is again a `∗`. Although each of our current Arrangers eliminates overlaps completely, this extension point is required. On the one hand, the separation of Arrangers makes sense because we are now able to assign a Comparator to an Arranger directly. These two components only work correctly when they are employed together. Assigning the Comparator to a `composer`, i.e. to the `strategy`, which has more than just one Arranger might yield unwanted composition results. On the other hand, as mentioned, not only the diagram can be arranged but even the model. This requires at least the combination of the `ModelArranger` and one Arranger for graphical fragment diagrams.

```java
// => Viewpoints XML Use Case
activate org.reuseware.lib.systems.usecase.uml activate org.reuseware.lib.systems.participation.uml
activate org.reuseware.lib.systems.participation.uml activate org.reuseware.lib.systems.participation.uml
```

Figure 5.8: Activator and CompositionStrategy

Figure 5.8 shows a part of the activator file `concern_dimensions.rex_activator` where single concerns can be (de-)activated. Additionally, we can determine the layout composition strategy that will be applied to our diagrams. The displayed part shows the concerns `usecase` and `participation` and the `CompositionLanguage` (cl) the target information stems from. As can be seen, OpenOffice document is the `CompositionLanguage` for the usecase concern as well as for the participation concern. As mentioned in Section 3.2, the additional actor `Hall` is specified in another document which requires the activation of the second participation concern with the `UseCaseInvariant` composition language.

The `variant` expression determines the resulting language of a composition. This specifies which components are to be used for enrichment of the composition program. `usecase.uml` requires the use case elements depicted in the Figures 3.7 and 3.8. As result of these concerns’ composition a use case model with use case and actors is yielded. The layout language of this example is TOPCASED. Determining the `variant` already existed in Reuseware.

The `strategy` sets the string value for the layout composition strategy from the plugin.xml. We provide some strategies but a developer may implement more Arrangers and Comparators and compose strategies that support the different diagram categories better. Depending on the resulting diagram determined by the `variant`, the layout composition has to be adapted which is why we had to introduce the `strategy`.
Chapter 6

Evaluation of LaCoMe

In the previous chapters we presented the basis of our work and how we implemented our approach. We discussed the reasons for the implementation and introduced the components we added to our framework. Since we worked at this layout adjustment framework for almost one year now we had to test and evaluate it in order to find out if it really meets the goals we set.

What we wanted to create was an extension to Reuseware [GDa] which performs a rearrangement of composed fragment diagrams with respect to the goals of the Mental Map [ELMS91, MELS91]. These goals are:

Goal 1: disjointness of nodes

Goal 2: keep the neighborhood relationship of the nodes

Goal 3: compact design

In our previous work [Gau09, JG09] we presented examples where different fragment diagrams of the same type were composed. All the fragment diagrams had a graphical syntax. Now, however, we have an example where diagrams are transformed. We had to extend our approach in order to be able to transform a diagram of a certain type into one of another type while still preserving the mental map. Not only do we have different types of diagrams but we deal with both textual and graphical layout languages. Through this extension of LaCoMe, users have the opportunity to assign different layout composition strategies to the different concerns in a MDSD process. These strategies are meant to preserve the mental map in a way that a certain element of a fragment is displayed at the same location or at least with the same neighbors in the different types of diagrams.

What we wanted to achieve by this evaluation is a quality check of our work. Furthermore, we wanted to get suggestions for improving the usability of LaCoMe.

In Appendix A we present the survey sheets we handed out to our probands. Eight employees of the chair of software engineering of the TU Dresden participated in that evaluation. Some of them are familiar with the Reuseware Composition Framework but none ever used LaCoMe before. Additionally, one external IT specialist and a student helped to improve LaCoMe by answering our questions.
Therefore, we provided an Eclipse installation and a workspace containing all required implementations and the TicketShop example. Furthermore, we developed a screen cast and a short presentation for introduction purposes. All is available online at http://reuseware.org/downloads/lacome.

As shown in Appendix A, we posed nine questions to which the probands were to answer by giving school marks. The additional three questions were to answer with free text. In order to offer space for further remarks the probands would like to give, we added question thirteen.

In the following, we not only interpret the results of the survey sheets but we additionally evaluate LaCoMe ourselves. Firstly, in Section 6.1.1, we have a look at the results of the manageability questions 1 to 4. Afterward, in Section 6.1.2, the marks for the questions 5 to 8 are interpreted. This section is about the results of the layout transformation. Thirdly, in Section 6.1.3, our approach is evaluated and we interpret the given marks and present the remarks as well as the suggestions for improvement the probands gave. Lastly, in Section 6.2, the additional comments of the probands (Question 13) are presented.

6.1 Evaluation results

This section presents the evaluation results of LaCoMe. Since we have three survey sections, we interpret them separately.

6.1.1 LaCoMe manageability

The first four questions were to evaluate the manageability of LaCoMe. We prepared an exemplary application which was a reduced version of the example from Section 3.2. The concerns usecase.uml,ucinv,valueflow and uml.class were activated and the probands had to add the layout composition strategies. In our presentation of the example application one proband asked if it was possible to combine different strategies by some syntax token like AND. Since we intend to prepare generic strategies by allowing many Arrangers, the composition of strategies would not make sense.

![Figure 6.1: (a) List of strategies (b) empty strategy tag](image)

In this section of the survey we wanted to know whether the probands were able to handle the given example. It would be no representative survey if the probands were
6.1. EVALUATION RESULTS

not able to execute the given tasks or do not understand the functioning of the application. As we presented in Section 5.2.8 users can (de-)activate concerns. However, in order not to distract the users, we asked them not to make use of this option but only use a set of strategies we prepared. What had to be done was copying. Figure 6.1(a) presents the list of prepared layout composition strategies we provided. The empty tag for the strategy is shown in Figure 6.1(b).

![Figure 6.1(a)](image)

**Figure 6.1(a):** List of prepared layout composition strategies

Figure 6.2 depicts the `BookTicket.usecase.uml` diagram with the default layout adjustment strategy applied. As stated, this strategy is automatically executed if none has been specified by the user. There is no Arranger specified in this strategy but neither are `deactivateMergers` elements assigned. Thus, if `canMerge` returns true for any of the implemented Mergers, the diagram merge is executed.

Since we have six layout composition strategies and—in the usecase.uml example—three strategy tags, the probands had a lot of possible combinations they could test. In order to reduce this amount we only asked to use strategies that are likely to yield readable results (cf. Appendix A).

![Figure 6.2](image)

**Figure 6.2:** Use case diagram without arrangement

Figure 6.3 shows one of many possible combinations. For this example we only applied `HorizontalSorting`.

![Figure 6.3](image)

**Figure 6.3:** Use case diagram with horizontal sorting applied

Table 6.1 presents the results of Questions 1 to 4. This result shows that many users had problems using the provided example. It seemed to be hard for the probands to interact with the software intuitively. While we even thought about providing less information to avoid overloading the users we were told that our example was too little documented for inexperienced users. Thus, the examples’ documentation got an average of 3.6 and for the first question we got a 2.8. However, this seems to be a
problem of Reuseware itself instead of one of LaCoMe’s since we did not build an own example to make the example fit our purpose but we took the example from Section 3.2 and made LaCoMe work for it.

### 6.1.2 Transformation results

After we evaluated the usability of the given implementation in the last section we want to know how the transformation results appeal to the users. The probands were to evaluate the diagrams that are yielded when a transformation was executed. Again we posed four questions that were to be answered by giving school marks.

The purpose was to determine whether the users recognize the fragment diagrams in the composed results. Therefore, we provided the fragment diagrams on the survey sheets (cf. Appendix A). We now present the diagrams that should be yielded when LaCoMe was used as intended.

![Figure 6.4: (a) Order in BookTicket.odt (b) order in BookTicket.ucinv](image)

The layout management, or layout preservation, for the diagrams was quite difficult since, being from different sources, the target information had to be combined in order to preserve the neighborhood relationship. In [Gau09] this target information was gathered from just one source, the composition program. However, in the motivating example this work is based on we had to gather layout information from both an OpenOffice document and a UseCaseInvariant document for the use case and participation concerns.

Figure 6.4 depicts the sources for the required layout information. The implemented TargetDiagramInformationProviders for these languages then extract that information. In Figure 6.4(a) the values for the three actors Customer, Clerk, and Bank are highlighted. This list index of the SpanType list they were added to now
6.1. EVALUATION RESULTS

serves as the y value of the targetBounds. Values for the fourth actor Hall are gathered using the EMFTextTargetInformationProvider since they are in the UseCaseInvariant document depicted in Figure 6.4(b). As shown in this figure, values for different actors clash, e.g. in Figure 6.4(a), (1) is the value for Customer whereas the same value is assigned to Bank in the UseCaseInvariant document. However, in our example the first source is the dominant one and only the counter actors specified in the UseCaseInvariant document get the assigned value (cf. Section 3.2 and Listing 3.3).

![Figure 6.5: (a) UseCaseInvariant result without (b) and with layout adjustment](image)

Firstly, we present the layout composition result for the variant ucinv. This means, that the fragments for enriching the composition program are UseCaseInvariant fragments. Since both, the sources of the information and the transformation result are textual languages, this example best clarifies our aims. Figure 6.5(a) depicts the composed text when no layout adjustment is applied. What happened here was the composition of two separate models. The three actors from the ODT were composed first in the order of their appearance and after that, Hall was simply added to the model. However, in Figure 6.5(b) the ModelArrangement was applied for the concerns. Again the documents are in the same composition order but now the target values are taken into consideration. Hence, the first three actors are still in the same order but with Hall being added afterward, Clerk and Bank are moved one line in order to make room for Hall.

In Figure 6.2 the BookTicket use case diagram is displayed. Would we drag the topmost node off the stack of actors we could see that it is Hall. Though being a different model, i.e. usecase.uml, the order of the fragments is the same. Once these fragments are sorted properly, as can be seen in Figure 6.3, they have the same order they have in the UseCaseInvariant document. This is one of our goals; preserving the layout in a MDSD process. Although this is not so relevant for our small models in this example, it may be for big ones. Users are now able to at least make an educated guess if not even know exactly where the fragments are when they switch from one representation to the next.

What users additionally require knowledge about is the layout composition strategy that was applied to the new composition result. Not knowing about this makes the rearrangement redundant since users are again lost because they can only guess where a certain fragment is. And there is a big gap between an educated guess and guessing without background knowledge. Therefore, we categorized diagrams regarding their layout restrictions in Section 4.2. There should be at least one layout strategy for

---

1 We do not claim to be exhaustive here.
Table 6.2: Transformation results

| Question 5 | 2 | 1 | 2 | 3 | 1 |
| Question 6 | 5 | 1 | 1 | 1 |
| Question 7 | 1 | 2 | 1 |  | 2 |
| Question 8 | 1 | 3 | 2 |

The users’ satisfaction with the diagrams is presented in Table 6.2. Due to the rather satisfactory grading (around 3) of the usability of LaCoMe in our example, the evaluation of the composition results are not so good either. We wanted to know to which extent the composed diagrams met the expectations of our probands. Although Question 5 was marked with an average of 4, the second question, Question 6, about the recognition of the source diagrams in the composition was marked with 2.8. Hence we assume that, though there are some drawbacks in finding the right strategy, we managed to provide a solution which at least makes the diagrams readable in so far as its elements are easily recognizable even by non-experienced users. Question 7 is marked with a 3.0 and hence, we have to admit that it will be inevitable for the future to develop more sophisticated strategies for the different diagram categories. Unfortunately, we did not provide the rules a diagram has to obey in order to meet the goals of the mental map. Therefore, only few probands were able to answer question 8. Those who did marked the results with an average of 3.5 because, as we were informed later they assumed that, besides removing overlaps of the nodes, we wanted to avoid overlaps of edges and labels too. However, this was not intended but it could be subject to future work.

Altogether, the transformation results are satisfying for our probands with some drawbacks caused by the missing background knowledge of the users. We come back to this section when we interpret the additional comments we received.

6.1.3 Layout adjustment approach

In this section we present the evaluation of our approach in general. We wanted to know whether it was a good approach for our aims and if the probands have suggestions to improve it. In this part of the survey, we offered a lot of free space for the probands to fill with suggestions and one more question to mark the prototype. Although we cannot discuss all suggestions, some are presented and interpreted here.
6.1. EVALUATION RESULTS

Appendix B provides the list of all answers and recommendations we received.

As mentioned, our intention was the preservation of layout information in a MDSD process. Furthermore, the mental map was in focus and therefore, the readability of graphical diagrams. The latter has already been a goal in [Gau09] and we proposed a layout adjustment approach which enabled the user to choose between different arrangement implementations in order to remove overlaps (Goal 1), preserve the neighborhood relationship (Goal 2), and create a compact layout (Goal 3) in composed diagrams. However, in this work we extended this layout adjustment framework and created LaCoMe by which layout information should be preserved not only in a composition but in a transformation as well. Since the basic layout of diagrams or the representation of the underlying models' elements within that diagram changes in the different types, we had to find a categorization of the different types. Moreover, we had to provide the option to employ adapted adjustment strategies for the diagram types.

Figure 6.6: ValueFlow participation concern horizontally sorted

Figure 6.6 again depicts the transformation result of BookTicket.odt and BookTicket.ucinv (cf. Figure 6.4). However, this time the composition program is enriched with fragments which have a GMF syntax. This view and this (ValueFlow) language are for illustrating the flow of values in the software. Each actor has some values he has to give and some he has to take. This passing has a specific order which is defined in ValueFlow.

In Figure 6.7 the fragments for enrichment are shown. On the left hand side, i.e. Figure 6.7(a), the diagram for the participation concern is shown. It corresponds to the actor node in a usecase.uml diagram (cf. Figure 6.3). Of course there is again a diagram for the use case concern but it is empty in this case and we therefore do not show it here. On the right hand side, i.e. Figure 6.7(b), the fragment for the passing of values is depicted. As can be seen this is a containment design or, as we call it, a 3D design. Having such a relationship requires distinct arrangement strategies as we developed them.

Figure 6.8 depicts the result with all concerns activated. In this example we applied a horizontal sorting for the participation concerns while the value exchange concern is sorted vertically. As can be seen, both the participation and the value exchange elements do not overlap for our choice of strategies. Therefore, the approach
Figure 6.7: (a) Participation fragment (b) GiveState (GS) and TakeState (TS) for value exchange

Figure 6.8: All ValueFlow concerns with different strategies

of the strategies fits the purpose. However, choosing freely may be a too difficult task for users since there are too many combinations that can be tested. Sparing this time consuming task by providing a recommendation for certain diagram types may improve the usability of LaCoMe.

Furthermore, developers have to make up their minds about how they lay out the fragments that are meant to enrich the composition program. As can be seen in Figure 6.8, there is a scrollbar on the right hand side of the Customer actor. Had we shown the whole diagram instead of this snippet there would be even more scrollbars. These are caused by the size of the fragment in Figure 6.7(b) and the diagram composition strategy. Since, by aligning the fragments vertically, we want to remove overlaps, their height determines how far they have to be moved. If the PLACEHOLDER compartment of the value passing fragment is too tall the moved elements’ position exceeds the participation elements’ bounds and causes scrollbars. In a different syntax than GMF it can even cause the element to disappear. Beside the adaptation of the enriching fragments, there is the option to resize the result, i.e. adapt the actor element, in order to overcome this flaw.

The answers to the ninth question are summed up in Table 6.3. Again some
6.1. EVALUATION RESULTS

<table>
<thead>
<tr>
<th>Question 9</th>
<th>1</th>
<th>1</th>
<th>3</th>
<th>1</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
</table>

Table 6.3: Approach

probands were rather satisfied while others did not like the approach at all. Mostly, those who gave bad marks did not understand the example and hence did not rate other questions at all. However, there were constructive remarks for the Questions 10 to 12 which we now discuss.

**Question 10** concerned the recommendations of our probands for LaCoMe to enhance the support of the different characteristics of diagram categories. As can be seen from the given answers, the users were of contrary opinion about how the layout adjustment should be applied. On the one hand it was suggested that users should be provided the option to freely choose the layout strategy for a certain diagram category, save that choice and apply it to the different diagrams of that category. On the other hand, another proband said we should just disable or prohibit inappropriate strategies. We think there are advantages and disadvantages for both these suggestions. As explained in [Gau09], we want to have an interactive layout adjustment which implies that the user is not bereaved of the option to freely choose the layout strategy she wants. With this we argue in favor of the first suggestion. However, providing too many strategies—one for every category—distracts the user and makes the choice a time consuming task. The second part of the first answer, i.e. to save a choice and apply it again for the same category, equals the suggestion to provide defaults for the strategies. This is quite a good thing and should be done once satisfying layout strategies are found. However, we did not concentrate on the realization of that but just wanted to provide the framework which enables the implementation of that option in coherence with the mental map preservation. Some answers to this question concerned the layout arrangers. Of course there are more sophisticated adjustment methods than sorting horizontally or vertically. A force-directed approach however, as proposed by a proband, adds an equal amount of space between the model elements. It may be hard to conceive since we have textual target information, i.e. the OpenOffice document, but we want users to have the freedom to arrange the fragments freely on the canvas. After the transformation the space between the different elements should be as the user wanted it. We discussed this in [Gau09] where we had access to the physical composition program. However, adding space between two elements is necessary and we did not implement this in [Gau09]. More precisely, we did not gather the difference value of two succeeding fragment illustrations of the composition program in order to add this to the composed fragments. In our current work, i.e. the **MultiHorizontalArranger** and the **MultiVerticalArranger**, some space is added to tell one fragment from another.

**Question 11** deals with the diagram characteristics the probands think are not sufficiently supported. It was mentioned that the sub states in the ValueFlow example are arranged in a way that requires scrolling. This flaw is depicted in Figure 6.8 and we pointed out some solutions to it. Furthermore, a proband states that there are many strategy combinations which do not remove overlaps. We can only guess that he is referring to overlapping labels in particular because, put aside the UniformScaling,
all Arrangers recursively check for overlaps and do not stop rearranging until these overlaps are removed. However, label overlaps are not in focus of this work but should be in the future. This requires a separate approach for the labels since removing their overlaps for improving the readability must not interfere with the arrangement of the fragments. If it did, the mental map could be destroyed. One user mentioned the lack of support for activity diagrams with swimlanes. We have to object that since the swim lanes are similar to the actor fragments in the ValueFlow example and the activities can be mapped to the sub states. Therefore, the support is implemented and with a combination of the HorizontalSorting—the one without the additional gap between elements—for the swim lanes and VerticalSorting for the diagram elements should yield a satisfying result.

**Figure 6.9: Non-user-friendly order of objects in a sequence diagram**

**Question 12.** Since we had only some diagram types in our example more test cases have to be examined. We wanted to know if the probands would suggest any diagram types we should pay special attention to. One user recommended sequence diagrams as a type that should be subject to future work. As a matter of fact we are even now able to adjust the arrangement of these charts. We are able to align the object along the x axis and the messages can be sorted vertically. However, messages should have a certain order and overlaps with lifelines should be avoided. Figure 6.9 depicts a violation of this rule caused by the displacement of an object. In Figure 4.7 Bank is on the right hand side of ACM Machine which would avoid the overlap of the messages check Card and valid with the User lifeline. Moreover, the first message of this diagram—inser Card—is sent by the user object. Hence, User should be on the left hand side of this illustration. A second suggestion was that we need to concentrate on complex inheritance trees. LaCoMe supports this kind of diagrams in so far that the three goals of the mental map are supported. One could assume that a combination of vertical sorting for the layers and horizontal sorting for the elements within one layer yields a good result but it has not been tested yet.
6.2 Additional comments of the probands

Question 13 of our survey is an additional one where the probands were asked to write additional comments they found were not covered by the Questions 1 to 12. Firstly, users recommended to improve the survey itself along with the evaluation. One mentioned that the survey should differentiate between the different languages of our example. This separation allows the distinct rating of LaCoMe’s support for the different compositions. Hence, a more fine-grained evaluation is possible and the goals for future work can be better set. Another suggestion was to prepare composed diagrams and present these to the probands instead of leaving the choice and application of the strategy to the users. Reflecting on the results of the evaluation and the problems the testers had with our example we come to the conclusion that it is better to use both approaches. On the one hand we do not want to bereave the users of the free choice for a fitting strategy. On the other hand, however, the lack of background knowledge and the complex example led to difficulties in the handling of LaCoMe and often no transformations could be produced. Therefore, exemplary results should have been provided which show our aims.

In addition to the free space we provided for the answers of Question 13 the probands added annotations to some of the Questions 1-9. Since these remarks are sometimes very helpful for determining why the particular tester rated in a certain way, we now have a look at some of these annotations. As a start, one user stated that our example is very complex since the combination of a textual language and layout adjustment is hard to grasp. Actually, we have to agree but nevertheless there are textual use case descriptions in software development and our intention was to search for an option to preserve even the layout information that is not visible on first sight. However, what could assist the user in the comprehension of the software is a forecast of the solution. In the example from Section 3.2 this would imply that for each of the transformations in the activator file there would be this forecast which asks the user if the solution is satisfying. If it is not, the user is to interact with the software and rerun the adjustment. This would be a tedious task we want to spare the users. However, what is imaginable is the combination of such a forecast—or better an adjustment of the adjustment—with the default suggestion discussed earlier. We think that we could provide the preservation of the layout in so far that the neighborhood relationship is kept and that we use a default strategy for the diagram category. Having gathered the layout information of both target and source anyway, we are able to produce more variants of that diagram with different strategies applied. These could be provided for the user to choose from when the default is not satisfying. Another option which aims at improving the understanding of the example would be to provide more information about it. Starting off with the layout preservation between textual documents and graphical diagrams probably was not the best thing. We could have divided it into two parts by providing an example with graphical diagrams only at the beginning and introduce the text in a second step.

Question 8 was about whether or not the mental map principle was sufficiently supported, i.e. the Goals 1-3 are achieved, by LaCoMe. One tester annotated this question with the remark that this was only the case when the tool is used correctly and the target models’ constraints allow the support of the mental map. He suggested to employ more tooling. We agree to the point where he mentioned the correct use
of the tool. Assuming that he meant the use when the user is trained and has some background knowledge. However, this is right for every software on the market. If the usage of it is out of the user’s depth he requires a certain training. For us this would mean that a deeper introduction and additional documentation for LaCoMe is required. The second part of his comment where he mentioned that it depends on the target models’ constraints whether the mental map is supported or not has to be challenged. We doubt that there are any diagrams where overlapping fragments (Goal 1) are a volitional feature unless of course there is the 3D aspect involved. These overlaps, since they are intended and recognizable do not influence the layout adjustment in so far that they would violate the rules of the mental map. On the other hand Goal 2, i.e. the preservation of the neighborhood relationship, is one feature that can cause problems in the different diagram categories. We discussed this in the previous section an Figure 6.9 depicts the problem. However, for the mental map, this is less important than the removal of overlaps.

As to the results of the transformations one proband mentioned that there were no changes in the diagrams although he changed the strategy. We thought to have solved this problem already in [Gau09] since we had to deal with it back then too. However, if the diagram is opened from the fragment repository view and the strategy in the activator file is updated (and saved) the diagram is not redrawn. Only if the diagram is closed and opened again after the update, the changes are visible. However, this has to do with the update of the abstraction since the change is immediately taken over in the composition program and the target model. Therefore, the error is caused by the update function of the external graphical editor. Another possible cause for this problem is a typo of the user. We did not handle a wrongly typed strategy at the time the survey was done but just redirected a non-existent strategy, i.e. null or the empty string, to the default strategy which only merges the involved fragments. Thanks to the survey this flaw was detected and we implemented the check for undefined strategies. We think that a content assistant which provides only the available strategies is a helpful feature that has to be implemented for a better usability.

In Question 6 the probands were asked if they are able to find the original components again in the composed diagrams. One who marked this question with a 2 demanded to know what happens if the complexity of the examples is increased. Although we were not able to test it we assume that, with the knowledge of how a strategy works, users are able to find the components again even in complex diagrams. However, if we at least are able to preserve the layout of diagrams to a certain degree—which we did—this work contributes to the readability of composed diagrams. We have to admit that there are some constraints which do not allow such preservation of the layout but even so the mental map adjustment can be applied which improves the layout. In many cases however, we are able to keep the neighborhood relationship of fragments in the different diagrams and hence, the usability is increased because users know where to find original elements of their transformation.

To sum it up, we learned that, although there are some disadvantages in the usability of our example, most users were able to apply layout adjustment strategies to the different diagrams and the original components could be found again in the resulting diagrams. Since we did not create an example ourselves which is easy to use but got our requirements for layout preservation from an already existing example,
its usage was determined and we had to think of a way to introduce LaCoMe. Hence, it is comprehensible that our probands were not satisfied by the usability and recommended its improvement. In addition to this, the users proposed several adjustment approaches like a cost-based approach which should definitely be subject to future work in order to enhance the support of different diagram categories. Furthermore, the testers demanded to know whether LaCoMe is applicable if the software complexity is increased, i.e. the fragments consist of more elements. Although this remains subject to future work as well, we assume that it should be no problem to adjust our framework in a way that it can handle this new complexity. Finally we learned that, at the current state of development, LaCoMe does what it was designed for, i.e. preserve layout information in MDSD processes and adjust diagrams in a way that their elements obey the rules of the mental map. However, the support for diagram categories has to be improved.
Chapter 7

Conclusion

In this chapter we reflect on the work we did and on the goals we achieved. Furthermore, we point out the things that LaCoMe still lacks and whether or not these features are required and should be realized in future work.

As a start, i.e. in Section 7.1 we have a look at related work. Secondly, Section 7.2 discusses the advantages and disadvantages of the motivating example for LaCoMe. Thirdly, in Section 7.3, we refer to the diagram composition strategies, i.e. our solution of the given tasks. Afterward, in Section 7.4, we analyze the conformance of LaCoMe to the requirements of the layout preservation in MDSD processes. Finally, we conclude the thesis in Section 7.5 by giving suggestions for future work based on the analysis of the achieved goals and, of course, the drawbacks.

7.1 Related Work

In the field of human-computer interaction there are some projects dealing with the preservation of the mental map [DG02, EKLN03, PHG07]. These look at dynamic graphs that change over time, i.e. an endogenous transformation. In a model composition and transformation environment the graphs change over time, too. However, this is more complex since we deal with endogenous/horizontal and exogenous/vertical transformations at the same time like it is the case for Reuseware or the work of Pilgrim et al. [vP07, PVSGB08].

In their work, Pilgrim et al. [PVSGB08] tackle the problem of a monolithic transformation’s bad scalability and little reuse opportunities by decomposing it into several sub-transformations. The model elements which are involved in a transformation can be traced back to earlier models by keeping traceability models and relations to other models in the chain. These traceability models not only help to find the transformed elements in the next intermediate model but they also assist in preserving the mental map. The UniTI tooling presented in [PVSGB08] derives a target notation model from the source notation model which is assigned to the first domain model of the chain. However, [PVSGB08] says nothing about how possible overlaps are removed in UniTI. With LaCoMe and Reuseware we are able to transform source domain models into target domain models and preserve the layout information from
There is no categorization of diagrams with respect to their layout and their layout restrictions in the literature. The UML superstructure [Gro09] defines two categories: behavior and structure diagrams. This categorization, however, does not help when it comes to constraints in the layout of diagrams which is why we had to come up with our own. Siebenhaller and Kaufmann [SK06] present a layout approach for activity diagrams and defined aesthetics for it but there is no general categorization to be found.

Many research efforts have been put into preserving the mental map in endogenous systems but there are only some which provide support for exogenous transformations. Furthermore, there are no layout management actions like the one we discussed in this work, i.e. the preservation of layout information in a whole MDSD process.

7.2 Motivating Example

The example we based this work on was Multi-Dimensional Separation of Concerns [OT00] realized with Reuseware. As discussed in Section 3.2, this implementation allows for the employment of different composition languages and component models. Furthermore, we were able to create different views on the software as it is common in a model-driven software development process. At the beginning, there was a use case description which was to be transformed into different representations, each showing another aspect of the software and finally the executable Java implementation is automatically derived.

However, according to the probands of our evaluation, it was hard to grasp the functioning of the software without thorough documentation. Actually, it is not easy to think about graphical layout when the basis is a text document. Nevertheless, once users are trained how to use the example and LaCoMe they should be able to work with it. Were we to do another evaluation, we would provide another example which is easier to understand.

7.3 Diagram composition strategies

With our motivating example came a variety of different software models which lead us to the question whether there are common rules according to which diagrams of these models are drawn. Since we did not find such a categorization but found it helpful to have an overview of the possible arrangements and layout restrictions of diagrams, we had to come up with diagram categories ourselves. Into these categories we sorted diagrams that are common in today’s software development environment and some more well known types.

Most of the diagram types we found can be arranged with the diagram composition strategies. We decided to refactor the layout composition framework from [Gau09] since we noticed that the requirements have changed once we do not just deal with diagrams that could be arranged freely on the canvas but also with the ones that are restricted in their layout. Therefore, the Arranger has been removed from the attribute list of a strategy to become a separate extension point. Additionally, the
7.4. Layout preservation in MDSD

Composer is not an attribute of the strategy any more but is now assigned to each Arranger. We also learned that some models may require more than one arrangement which is why we decided to allow multiple Arrangers.

Having this altered approach to layout arrangement does not change the goals we set in [Gau09]. We still support the preservation of the mental map since it is one major usability feature. Improving the readability of diagrams was our main task and, although UniformScaling is not the best adjustment method, we achieved it with the implementation of our strategies. However, there are some disadvantages in the arrangement implementations but finding new and more suitable layout adjustments can be subject to future work. What we did was building the framework which enables future developers to easily introduce new strategies and languages.

7.4 Layout preservation in MDSD

The new contribution of this work was the management of layout information in a model transformation environment. In MDSD processes, many different models are employed where each has a different representation. Some even have restrictions regarding the layout of the representation. On the one hand, these restrictions are bound to the semantics of the software and on the other hand, no conflicts arise when the layout is changed. Hence, we had to find a way to cover both types. Our aim was to preserve layout information like the neighborhood relationship of elements when the model is transformed.

To a certain extent we were able to meet the goals we set. Semantically restricted diagrams—whereby diagram includes here also text—like, for example, Java source code, has to obey to semantic rules in order to produce executable software.

With our strategy concept we managed to preserve the layout of both semantically restricted and semantically not restricted diagrams. However, in some cases we could not preserve all the information there was since the constraints did not allow it.

7.5 Future Tasks

There are some things that have to be done to improve the usability and the effects of LaCoMe.

As a start, the table in Section 4.2 has to be extended since we do not think that we found all diagram categories. Having a complete or at least more detailed overview of the diagram types would help to improve the strategies because more restrictions or other commonalities can be found. These attributes could also assist in developing a more fine-grained categorization. One more example could be a class diagram with design patterns. These could restrict the layout and hence, new strategies have to be developed.

Secondly we suggest to improve the tooling LaCoMe was integrated in. Until now the update functionality of the external editors does not work as intended. Although the model and the layout information is updated, the diagrams do not change, when
they are open. Consequentially, users assume that nothing changed which causes confusion.

As stated earlier, with LaCoMe we provide the framework to integrate multiple layout languages and adjustment strategies. However, finding appropriate Arrangers for the different diagram types still remains subject to future work. Although we implemented Arrangers to sort horizontally and vertically, which allows, especially when combined, for the arrangement of many diagrams, better methods may be found. In their comments our probands already recommended two. In order to support the diagram types, one user suggested a cost-based approach where rules have to be created and every violation of a rule causes a penalty. The best adjustment method can then be chosen automatically. However, since this automation could reduce the users' influence, the parameters that can be set in the rules have to be chosen carefully. Another option that was recommended was a force-based approach. This has to be explored as intensively as the cost-based approach because assigning the same force to each edge or node in a diagram would certainly remove overlaps but it has to be combined with a mechanism to preserve the mental map. Otherwise, all nodes have the same distance which could cause the destruction of the mental map.

A suggestion one of the probands who did the evaluation gave us, could help to improve the usability of LaCoMe. He mentioned a forecast or preview of the result the application of a certain strategy yields. We discussed that in Section 6.2 and we think this would be a good option when combined with a default strategy for a diagram category. What we do not want is that the user is asked for every concern that is involved in the composition. This would be tedious since once the decision is made, the layout process has to be rerun in order to apply the changes. However, applying a default strategy and providing a graphical user interface where the strategy can be changed afterward would be an option. Of course, this GUI has to have a preview of the changed diagram.
Appendix A

Evaluation Sheets
Evaluation des Layout Composition Frameworks (LaCoMe)

Nachdem Ihnen das vorliegende Beispiel erklärt wurde, bitten wir uns einige Fragen hinsichtlich der Verwendbarkeit unserer Implementierung zu beantworten. Der folgende Fragebogen enthält sowohl Fragen, die mit Schulnoten (1 = sehr gut, ..., 6 = sehr schlecht) zu bewerten sind---bitte ankreuzen---als auch solche, in denen wir Sie bitten Ihre Meinung auszuformulieren. Bitte geben Sie diesen Evaluationsbogen bis spätestens 01. April 2010 bei Jendrik Johannes (Raum 2082) oder bei mir, Karsten Gaul, im Labor für Softwaretechnologie ab.


Handhabbarkeit

1. Intuitivität der Handhabung
2. Ist die Beispielanwendung verständlich?
3. Inwieweit eignet sich das Ihnen vorliegende Beispiel für die Evaluierung?
4. War die Einführung ausreichend, um Aufbau des und Umgang mit dem Tool zu verdeutlichen?

Ergebnis

5. Inwieweit entspricht das Ergebnis Ihren Erwartungen?
6. Wie gut finden Sie die einzelnen Originalkomponenten (siehe Handzettel) in den komponierten Modellen wieder?
7. Werden die Restriktionen unterschiedlicher Diagrammarten ausreichend berücksichtigt?
8. Wird das Prinzip der Mental Map ausreichend unterstützt?
Ansatz

9. Wie gut eignet sich der Ansatz der frei wählbaren Strategy für das angestrebte Ziel?

10. Welche Verbesserungen würden Sie vornehmen um die Eigenheiten von Diagrammarten besser unterstützen zu können?

11. Gibt es Diagrammeigenschaften, die Sie nicht ausreichend unterstützt sehen? Welche?

12. Wir planen natürlich noch weitere Testfälle. Welchen Diagrammarten sollten wir Ihrer Meinung nach besondere Aufmerksamkeit schenken?

Herzlichen Dank für Ihre Unterstützung!

13. Haben Sie noch zusätzliche Anmerkungen?
Handzettel zur Evaluation (1)

Dieser Handzettel präsentiert die jeweiligen Quellkomponenten für die Komposition und die möglichen Layout-Adjustment-Strategien. Die Zielinformationen, die zur Komposition notwendig sind, werden in anderen Komponenten gekapselt, die wir auf einem Zusatzblatt zeigen. Hier werden lediglich die URIs dieser Komponenten (z.B. org/reuseware/application/ticketshop/fragments/odt) verwendet.

Die Ergebnisse finden Sie im Fragment Repository unter den URIs org/reuseware/application/ticketshop/integrated/* Suchen Sie also nach dem Ergebnis für usecase.uml, werden Sie unter */integrated/usecase.uml fündig.


UseCase.UML

1 und 2: org/reuseware/application/ticketshop/fragments/odt
3: org/reuseware/application/ticketshop/fragments/ucinv

1. HorizontalSorting, HorizontalSorting2, VerticalSorting, VerticalSorting2, UniformScaling
2. HorizontalSorting, HorizontalSorting2, VerticalSorting, VerticalSorting2, (UniformScaling)
3. HorizontalSorting, HorizontalSorting2, VerticalSorting, VerticalSorting2, (UniformScaling)

UseCaseInvariant

1 und 2: org/reuseware/application/ticketshop/fragments/odt
3 und 4: org/reuseware/application/ticketshop/fragments/ucinv

1. -4. ModelArranger
**ValueFlow**

UseCase.valueflow  
Participant.valueflow  
Value.valueflow

1 und 2: org/reuseware/application/ticketshop/fragments/odt  
3 und 4: org/reuseware/application/ticketshop/fragments/ucinv  
5: org/reuseware/application/ticketshop/fragments/valueflow/*

1. HorizontalSorting, HorizontalSorting2, VerticalSorting, VerticalSorting2, UniformScaling  
2. HorizontalSorting, HorizontalSorting2, VerticalSorting, VerticalSorting2, (UniformScaling)  
3. HorizontalSorting, HorizontalSorting2, VerticalSorting, VerticalSorting2, (UniformScaling)  
4. (HorizontalSorting, HorizontalSorting2), VerticalSorting, VerticalSorting2, (UniformScaling)  
5. (HorizontalSorting, HorizontalSorting2), VerticalSorting, VerticalSorting2, (UniformScaling)

**Class.UML**

Class.class.uml  
Association.class.uml

1: org/reuseware/application/ticketshop/fragments/odt  
2 bis 5: org/reuseware/application/ticketshop/fragments/ucinv

1. HorizontalSorting, HorizontalSorting2, VerticalSorting, VerticalSorting2, UniformScaling  
2. HorizontalSorting, HorizontalSorting2, VerticalSorting, VerticalSorting2, UniformScaling  
3. HorizontalSorting, HorizontalSorting2, VerticalSorting, VerticalSorting2  
4. HorizontalSorting, HorizontalSorting2, VerticalSorting, VerticalSorting2

Es wird bei keiner der Sprachen empfohlen, nur UniformScaling zu verwenden.
Handzettel zur Evaluation (2)

Auf diesem Zusatzzettel präsentieren wir Ihnen die Komponenten, aus denen wir die Zielinformationen bekommen. Die jeweilige URI wird dann repräsentativ im Handzettel verwendet.

**OpenOffice Document (org/reuseware/application/ticketshop/fragments/odt)**

**UseCaseInvariant (org/reuseware/application/ticketshop/fragments/ucinv)**

**ValueFlow (org/reuseware/application/ticketshop/fragments/valueflow/BookTicket)**
ValueFlow (org/reuseware/application/ticketshop/fragments/valueflow/ChangeSeat)

Customer.valueflow

- something
- Seat_oldSeat
- Seat_newSeat
- Address_add
- Shipment_link1

Clerk.valueflow

- Agent
- Address_add
- Shipment_link2

Hall.valueflow

- something
- Seat_oldSeat
- Seat_newSeat


Appendix B

Survey answers

In this chapter we present the answers probands gave to the questions 10 to 13.

B.1 Question 10

• ask the user which strategy she prefers for a certain diagram category and save this choice for future layout arrangement
• insert gap between graphical model elements
• HorizontalSorting: gap with respect to label width of a connector
• force-directed rules for diagrams could yield more natural pictures
• despite strategies have been inserted which do not exist, the elements should be displayed
• cost-based approach with penalties if certain constraints are violated
• dynamic approach suggesting appropriate strategies which can be activated via the UI might be better than the configuration-based approach with regard to usability
• defaults for certain diagrams
• interactive graphical editor with live re-layouting
• prohibit inappropriate strategies for diagrams

B.2 Question 11

• labels overlap
• activity diagrams → SwimLanes
• state automaton: sub states in Valueflow are arranged in a way which requires scrolling. One should stay within the borders of the parent state.
• only use case diagram worked. the other diagrams were not readable anymore or the change was not apparent.

• compartments in the ValueFlow example not big enough which requires scrolling

• there are a lot of combinations which do not remove the overlaps although the approach is fit for this task

B.3 Question 12

• sequence diagrams, activity diagrams, complex inheritance trees

• class diagrams

• diagram types with fixed layout constraints in the concrete syntax, i.e. spacial alignment relevant for semantics

B.4 Question 13

• unfortunately, the survey does not differentiate between the single examples

• models in the workspace and on the survey sheets named differently

• overlaps of connections between elements

• the evaluation of already adjusted diagrams might be better
# List of Tables

3.1 Transformation dimensions from [MCG05] .......................... 18

4.1 diagram layout restrictions ........................................... 31
4.2 categories of diagram layout restrictions ......................... 34

6.1 LaCoMe manageability .................................................. 60
6.2 Transformation results ................................................ 62
6.3 Approach ................................................................. 65
Listings

2.1 canMerge(FragmentDiagram) of GMFMerger .......................... 14
2.2 DiagramComposer ................................................. 14
3.1 fragment role 'UseCase' definition .............................. 23
3.2 odt.rex for the composition language .......................... 24
3.3 ucinv.rex for the composition language ........................ 25
4.1 sample Java code .................................................. 34
5.1 FragmentDiagram .................................................. 37
5.2 recalculation due to measurement unit in GMFMerger ....... 38
5.3 composeDiagrams(List<FragmentDiagram>, FragmentDiagram, Tra-
  ceProvider) of CompositionDiagramUtil .......................... 39
5.4 initDiagramCompositionStrategy() of CompositionDiagramUtil .. 40
5.5 TraceProvider interface .......................................... 42
5.6 DiagramCompositionStrategy ..................................... 42
5.7 Method findOdt(EObject) of ODTTip ............................ 44
5.8 Method findTarget(EObject) of EMFTextTip ...................... 46
5.9 Method provideBounds(FragmentDiagram) of EMFTextSip ...... 47
5.10 Method castToTextResource(Resource) of EMFTextSip ........ 48
5.11 Method mergeEMFText(EObject, EObject, int) of EMFTextMerger 48
5.12 Method getRemovedFromSetting(EObject) ...................... 50
5.13 Method isOverlap(List<FragmentDiagram>) of MultiHorizontalArranger 51
5.14 Method sortVertically(List<FragmentDiagram>) ................. 53
5.15 DiagramCompositionStrategy .................................... 55
5.16 check for deactivateMerger in CompositionDiagramUtil ...... 55
# List of Figures

2.1 Reuseware metamodels ........................................ 6
2.2 Fragment ....................................................... 7
2.3 Reuseware Extension components ............................. 10
2.4 CompositionDiagramUtil tasks ............................... 12
2.5 The parts of the diagram composition. (a) source and target information (b) composed diagram ........................ 12
2.6 Difference between FragmentDiagrams (background) and their illustrations in the composition program ...................... 13
2.7 (a) Composition program (b) Composed diagram with HorizontalAr- ranger applied .............................................. 15

3.1 (a) Monolithic transformation (b) transformation in Reuseware ...... 18
3.2 composition system requirements ................................ 19
3.3 BookTicket.odt .................................................. 20
3.4 Automatically derived composition program Main.fc ............ 20
3.5 Transformation and Composition System ........................ 21
3.6 (a) Transformation (b) Enrichment ................................ 22
3.7 (a) UseCase composition (b) Participant and UseCase Composition ........................................... 23
3.8 (a) BookTicket.ucinv (b) Participant and UseCase Composition ..... 25

4.1 Semantic metamodel and layout language metamodel ............ 28
4.2 A diagram of a model ............................................ 28
4.3 Graphical model of a composition program ...................... 29
4.4 Constellation Diagram with o restriction ........................ 30
4.5 UML Class Diagram ............................................. 31
4.6 UML Activity Diagram .......................................... 32
4.7 UML Sequence Diagram ........................................ 32
4.8 UML Use Case Diagram ......................................... 33

5.1 LaCoMe structure ................................................ 36
5.2 Structure of content.xml ........................................ 44
5.3 The getDerivedFrom relation in Reuseware ..................... 45
5.4 getRemovedFromSetting() ....................................... 48
5.5 (a) Overlap with bounding box (b) adjusted diagram .................. 50
5.6 Overlap check for fragment diagrams’ width ..................... 53
5.7 (a) Diagram before adjustment (b) incomplete adjustment (c) complete adjustment ......................................... 54
5.8 Activator and CompositionStrategy .............................. 56
6.1 (a) List of strategies (b) empty strategy tag .................................. 58
6.2 Use case diagram without arrangement ........................................... 59
6.3 Use case diagram with horizontal sorting applied ............................. 59
6.4 (a) Order in BookTicket.odt (b) order in BookTicket.ucinv ............... 60
6.5 (a) UseCaseInvariant result without (b) and with layout adjustment .... 61
6.6 ValueFlow participation concern horizontally sorted .......................... 63
6.7 (a) Participation fragment (b) GiveState (GS) and TakeState (TS) for 
  value exchange ................................................................. 64
6.8 All ValueFlow concerns with different strategies .............................. 64
6.9 Non-user-friendly order of objects in a sequence diagram .................. 66
Bibliography


Confirmation

I confirm that I independently prepared the thesis and that I used only the references and auxiliary means indicated in the thesis.

Dresden, April 29, 2010