Minor Thesis

Composition of layout information in Model-Driven Development

submitted by

Karsten Gaul

born 21.12.1982 in Eisenhüttenstadt

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

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

Submitted July 31, 2009
Aufgabenstellung

Komposition von Layoutinformationen in der modellgetriebenen Softwareentwicklung


Die Einarbeitung des Themas beinhaltet folgende Schwerpunkte:

- Einarbeitung in existierende Arbeit zur Repräsentation von Diagramminformationen (insbesondere der UML-DI Standard, und das GMF Notation Metamodel)
- Einarbeitung in das Reuseware Composition Framework und dessen Prinzipien der Modellkomposition
- Analyse existierender Ansätze zur Komposition von Layoutinformationen mit Hinblick auf Eignung für Reuseware
- Weiterentwicklung dieser Ansätze zum Einsatz in Reuseware
- Umsetzung dieser Ansätze als Erweiterung in Reuseware
- Evaluation der Ansätze an Beispielen
Contents

1 Introduction 1

2 Technologies 3
  2.1 Eclipse Modeling Framework (EMF) 3
  2.1.1 Metalevels and the Metaobject Facility (MOF) 4
  2.1.2 The Ecore Metamodel 5
  2.2 Graphical Modeling Framework (GMF) 6
  2.2.1 The notation metamodel 7
  2.3 Reuseware Composition Framework 8
     2.3.1 Fragment 9
     2.3.2 Composition 9

3 Layout Concepts 13
  3.1 Mental Map 13
  3.2 Layout Adjustment 14

4 Layout Algorithms 17
  4.1 Horizontal Sorting 17
  4.2 Uniform Scaling 19

5 Implementation 23
  5.1 Reuseware extensions overview 23
     5.1.1 Diagram Composer 24
     5.1.2 DiagramInformationProvider 25
     5.1.3 Comparator 25
     5.1.4 Arranger 26
     5.1.5 Merger 26
  5.2 The class CompositionDiagramUtil 26
  5.3 Reuseware extensions in detail 30
     5.3.1 SourceDiagramInformationProvider 30
     5.3.2 TargetDiagramInformationProvider 33
     5.3.3 Comparator 34
     5.3.4 Arranger 35
     5.3.5 Merger 39
6 Evaluation
   6.1 GMF example .......................................................... 43
   6.2 TOPCASED example .................................................. 49
       6.2.1 Class Diagrams ................................................. 50
       6.2.2 State Machines ............................................... 53

7 Conclusion
   7.1 Language-dependence and general approach ..................... 55
   7.2 Horizontal Sorting .................................................. 56
   7.3 Uniform Scaling ..................................................... 57
   7.4 Future Work .......................................................... 58
Chapter 1

Introduction

*Model-Driven Software Development* (MDSD) has emerged from a buzzword to a meaningful technology in software engineering. In this environment models mostly are visual graph representations, i.e. diagrams, drawn and used by software engineers. These diagrams are a powerful means of modern software development since, on the one hand, they bridge the gap between the software engineer and the client due to the high level of abstraction and, on the other hand, the developer is able to derive partial source code. In some cases even complete applications can be generated. Being a good basis for communication between customer and software engineer, models support the customer focused development and, thus, they are a good indicator for the acceptance of the product. The aspect that models can be input to code generators can reduce the duration and hence, the costs of the software development process. Achieving these goals, however, requires models that are easy to read and comprehend.

Furthermore, models in a model-driven software development process can be an easy way to specify, reuse and compose components. The *Reuseware Composition Framework* [11], currently under development at the chair of software technology of the TU Dresden, offers this kind of composition technique. *Fragment components*—or briefly *fragments*—can be developed by software engineers using some graphical editor. These fragments are stored and can later be reused for composition by selecting them from a *fragment repository* and linking the previously defined ports in another graphical editor. The result is a composed model of copies of the chosen fragments. Unfortunately, this composed diagram looks like the example shown in Figure 1.1.

This is just an easy example but if you imagine that you might have reused a fragment not only once but several times whereby each of them completely overlaps the other, this model can get quite large in depth. Diagrams that look like this cannot be used as a basis for communication since even the software developer is not able to see every detail. Hence, this diagram has to be rearranged by the user to see everything it contains. This, however, is a task you should spare the user if you want to provide user friendly tools.

Previous work on model composition only deals with the semantic elements of the models but the graphical layout has not been payed attention to yet. There are, however, mathematical algorithms for automatic layout creation but our work goes
beyond that. In Reuseware the user is able to arrange the chosen fragments in a graphical editor. By doing this he subconsciously stores the location of the fragments and maybe even the links between them. Hence, he creates his Mental Map [2] of the composed diagram. Keeping this Mental Map throughout the composition process is the major task of this work. What it means to preserve the Mental Map of the user will be described as well as the mechanisms that have to be implemented to realize this.

Simply put, the aims of this thesis are to make diagrams look nice and design them in a way they correspond to the user’s demands. But what is there to be considered? Can you tell from experience or past projects what is the best design? This thesis is based on the assumption that you can not. The sense of style is subjective but you may have also developed successive software which is restricted in its means for layout creation. Nevertheless, you might want a more successive product which is more user friendly. This can be achieved by implementing an adaptable layout creation approach.

This thesis is structured as follows. In Chapter 2 a brief summary of some basic requirements is given. Namely, these are the Eclipse Modeling Framework, the Reuseware Composition Framework and the Graphical Modeling Framework. In case you are already familiar with the latest versions of these frameworks you may want to skip that chapter. New concepts—at least for those who never dealt with layout creation before—will be introduced in Chapter 3. After that, i.e. in Chapter 4, two algorithms for layout adjustment will be presented. Putting it all together, Chapter 5 provides the solution to the given problems. In order to evaluate the implementation, Chapter 6 provides step by step examples for two component layout languages. Finally, a conclusion is made in Chapter 7 featuring a reflection on the process as well as future work.
Chapter 2
Technologies

As a start, it is inevitable to look at the basis this work is built on. Without the lecture of this chapter, following the rest of this thesis is hard for those who are not already familiar with EMF, Reuseware and GMF. This chapter, however, is a good repetition. Although the headlines might contain known terms there will be some more background information included. At first, that is in Section 2.1, we briefly introduce EMF and the Ecore Metamodel as well as EMOF. As an example for the user interface component of Reuseware, GMF is introduced in Section 2.2. All that knowledge is required to give an introduction to Reuseware and its functionality. This introduction is presented in Section 2.3.

2.1 Eclipse Modeling Framework (EMF)

EMF [3] is a framework built to assist software engineers in the software development process. Different software engineers, however, have different approaches to this process. There are those who like to work on a low level of abstraction, i.e. programming Java code, and there are others who prefer modeling in a high level graphical modeling language like UML and derive initial code from that. Both approaches are supported by EMF and you can even transform one representation into the other. Although modeling in UML might be seen as a redundant time consuming task by some programmers, it does have benefits. In a meeting with a customer, for example, these UML models can serve as a means for communication since customers usually are not that familiar with programming languages. Another upside is that the relations between interfaces and classes of your application are better illustrated. Apart from UML or Java there is yet another option supported by EMF. Maybe you neither like graphical modeling nor writing Java code but your knowledge about XML tags is very substantial. In that case, writing XML documents is your first approach to software development.

As depicted in Figure 2.1, EMF unifies all those three options and gives you the opportunity to generate the two missing representations from the one you created yourself.

An EMF model, however, consists of a variety of concepts whereof each is represented differently in Java, UML or XML. Class definitions, for example, are rep-
represented similarly in UML and Java but in XML they are complex type definitions. Attributes in UML are get() /set() method pairs in Java. Mapped to XML they are represented using nested element declarations. Thus, another model is needed that resides on a higher level and is able to define a common terminology for the three representations of the EMF model. Such a model is called a metamodel.

2.1.1 Metalevels and the Metaobject Facility (MOF)

<table>
<thead>
<tr>
<th>M3 (Metametamodel)</th>
<th>Modeling Concept</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Conceptual Level</td>
</tr>
<tr>
<td>M2 (Metamodel)</td>
<td>Class</td>
</tr>
<tr>
<td></td>
<td>Method</td>
</tr>
<tr>
<td>M1 (Model)</td>
<td>Attribute</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Building</td>
</tr>
<tr>
<td></td>
<td>void build()</td>
</tr>
<tr>
<td></td>
<td>Height</td>
</tr>
<tr>
<td>M0</td>
<td>tower</td>
</tr>
<tr>
<td></td>
<td>tower.build()</td>
</tr>
<tr>
<td></td>
<td>tower.height</td>
</tr>
</tbody>
</table>

Figure 2.2: Metalevels

Figure 2.2 depicts the metalevel hierarchy. To understand that hierarchy you need to know that “meta” is the greek word for “describing”. Starting at the level M0 the levels get more and more abstract and each one has concepts describing concepts of the previous level. While M0 is the actual implementation, i.e. software objects, M1 delivers the information about, what these software objects really are. This is then called the model of the application. Each application is written in a specific language, e.g. Java, C++ or C#, and the concepts of this language are defined on M2. Examples would be the structure of classes, methods or attributes. Finally, on level M3 which is the metametalevel the abstract syntax of a language is described.

The metalevel hierarchy is useful when a common language is used on M3. Such a language is the *Metaobject Facility (MOF)* [5]. MOF is a standard that was first
introduced in 1997 by the Object Management Group (OMG), a couple of enterprises interested in the development of standards relevant for the information engineering industry. The Unified Modeling Language (UML) [10], the Common Warehouse Model (CWM) [9] or the XML Metadata Interchange Format (XMI) [6] are just three of their research successes. MOF, however, resides on the metametalevel (M3) and is, therefore, a language description language. It is used to describe the abstract syntax of a language as well as to define logic constraints on the classes and their relations.

![Figure 2.3: MOF-compliant modelling](image)

In case software has to be adaptable to new languages, MOF might be the standard of choice. As shown in Figure 2.3 the syntax of a language can be modeled in a MOF-compliant way and once they have the same structure, different languages can be mapped to each other or they can be used in the same program.

### 2.1.2 The Ecore Metamodel

Due to its size MOF is not always the best choice. It is in particular difficult to provide a full implementation. With the release of MOF 2.0 in 2001 there is a reduced version called Essential Metaobject Facility (EMOF). Not only does this EMOF concentrate on the most important elements of MOF but it mostly is compatible to EMF.

Models in EMF are represented using the Ecore M3 language. As introduced above, a model which describes a model is called a metamodel. One could also use UML to describe EMF models but that would be like using MOF instead of EMOF. In EMF you just do not need the whole power provided by UML like modeling behavior using activity diagrams or sequence diagrams. The essential part for EMF is the option to create class diagrams which is why the Ecore metamodel has been developed. Ecore is based on EMOF and thus perfectly meets EMF’s demands. Figure 2.4 shows the Ecore hierarchy.

As stated, we are given three options to create an Ecore model of our language. We can either write XML documents or Java interfaces and generate the model from
them or we use UML. For the latter we simply require a graphical editor for Eclipse\(^1\).

The final question is how the generated model should be represented. Since it is a standard for serializing metadata—and an Ecore metamodel is metadata—XML Metadata Interchange (XMI) \([6]\) is used as the representation of the generated model.

### 2.2 Graphical Modeling Framework (GMF)

In the last section the Eclipse Modeling Framework was introduced. With its help, and given a metamodel for a language in either XML, UML or Java, it is possible to generate the other two forms. What has not been introduced yet but which is necessary to explain GMF is the Graphical Editing Framework (GEF) \([4]\). Since the details are not that important here we can simply say that GEF provides the means to create graphical editors. GMF is the technology to combine GEF and EMF. The data stored in EMF is displayed using graphical editors from GEF. Given that option we decided on GMF to be our composition layout language and one of two component layout languages (cf. Section 2.3). Basically, throughout this thesis we used GMF but we did some testing with TOPCASED \([17]\) (as another component layout language) as well.

Usually, you have to make a distinction between the data for the diagrams (i.e. icons, connections between these icons et cetera) and the semantic data. Users would normally say both are the same since, for placing shapes on the canvas and drawing connections between them, the semantic data plays a big role. The semantic data,

\(^1\)You can as well import or export from UML but this leaves you with the problem of the models being out of synch.
however, can be displayed in various editors and therefore should not store data about how it is displayed. Even more important in our case is that semantic data can be displayed several times in the same diagram. Just think about the option that one component can be reused multiple times in order to, for example, create the same behavior of the application in different circumstances. This would mean to store the diagram information—referred to as the notation information—several times, too. For that reason, the GMF notation meta-model was created which references the semantic data.

2.2.1 The notation metamodel

In Section 2.1 the metalevels were introduced. Now, these different stages of abstract models help to understand the concept of the notation metamodel. When we introduced Ecore we mentioned that there were three options to obtain a metamodel of a language. One of these options was UML and the only requirement was an editor for Eclipse. Since, by now, we have the means to create graphical editors (GEF) and the semantic data that needs some representation (EMF\(^2\)), we need some kind of bridge to connect them. This bridge is provided by the notation metamodel of GMF which sets the rules for this. Hence, it is a language to define concepts and rules for models which means it is a metamodel of level M2. Figure 2.5 shows the part of the notation metamodel that is interesting for this thesis. In our work we concentrate on the nodes of a diagram since their location and size is important for a proper layout. These nodes are of course parts of a diagram which itself is contained by a View. Within that View element there is the only link to the semantic model—a reference to EObject.

To show the difference between the semantic model and the layout Figure 2.6 depicts their different metalevel hierarchies. The reference to EObject shown in Figure

\(^2\)represented by Ecore
2.5 connects these two hierarchies. With the help of that connection, data like the location of the visualization of data from the semantic model can be stored. There has been some effort to develop a standard for representing that data. One of them is the UML Diagram Interchange (UML DI) [7].

### 2.3 Reuseware Composition Framework

In today’s software development, models play quite a big role since they abstract from the underlying code and, thus, give a better view on the structure of the software. This Model-Driven Development (MDD) aims at simplifying the software creation process since there are tools to derive source code from models.

There are, however, different approaches to MDD like, for example, the Model-Driven Architecture (MDA) [8]. This standard—of course it is one of OMG—is a proposal for a model refinement. Starting at a domain model which is referred to as the Computation Independent Model (CIM) a stepwise refinement towards the final model, the Platform-Specific Model (PSM), from which the source code can be derived is recommended. MDA, however, does not give any hints about how a model should look like. Even though you have an abstract representation of your program, you can get lost due to its complexity. Software for banks, for example, can get quite huge because there are so many use cases to be considered. Along with the size of the software, the size of its model increases. The MDA process—although being straightforward and starting at a very abstract level—therefore can get really confusing. An option to tackle this problem would be to reduce the size of models through modularization.

Our Reuseware Composition Framework [13] is a means to compose such model modules. Given a graphical user interface for both modeling the modules and modeling the composition it is possible to create components that can be reused later. In Figure 2.7 the Reuseware composition process is illustrated. $F1$ and $F2$ in this illustration are short for fragment component or just fragment.
2.3. REUSEWARE COMPOSITION FRAMEWORK

2.3.1 Fragment

Components in the Reuseware environment are called fragment components. These components can be incomplete and/or generic models which means that they are built for composition. Incomplete, in this case, means that there are some essential parts missing. Generic fragments can be incomplete too but there is at least one more part missing where another fragment can be put in.

To understand the composition concept of fragments the terms variation point and reference point have to be introduced. The first one is that part of a fragment that serves as a placeholder for other fragments. The latter, i.e. the reference point, gives a fragment a name so that it can be used in the composition. With variation and reference points being a rather abstract concept the implementation thereof might be easier to understand. Each fragment meant for reuse has to have some points to which other fragments can be bound to. In our case, this role is played by ports. So whenever the fragment is incomplete or generic, ports are implemented as addressable points. Due to their different tasks, however, there are three kinds of ports. Namely, these are the following:

- **Configuring Port**: These are used to change the composition result. There will not be any extension of the model which is why having only configuring ports in a composition will not be enough.

- **Contributing Port**: A contributing port offers additional elements required for a composition. Additionally, they can have some configuration functionality.

- **Receiving Port**: As a counterpart to contributing ports these ports need some information during the composition process. This results in an extended model after the composition. Furthermore, they can have configuration functionality.

2.3.2 Composition

Before being able to explain the composition process an essential distinction has to be made that is needed throughout the whole thesis. Since we are now familiar with the notion of a fragment we need to know that, in Reuseware, there are two abstractions of such a component. The first and detailed view on a fragment is the fragment developer viewpoint. A fragment developer has to model the fragment with all its details and, of course, the ports. The ports have to be assigned names by which they can be

---

3i.e. as variation or reference points
addressed later on during the composition. Once that is done we have a model in a specific component description language. Besides, any language defined by an Ecore metamodel can be used as component description language. In this thesis UML and a specific DSL for network configuration will be used. This DSL is based on the Common Information Model (CIM) [1][12][15]. The layout language for this example is GMF and in Section 6.1 we will present an example for that combination. The second example, presented in Section 6.2, uses UML for component description language and the layout language is TOPCASED. This example is taken from the Component Based Software Engineering (CBSE) course offered at the Technische Universität Dresden. In Figure 2.8(a) and 2.8(b) examples of such fragment diagrams are displayed. A fragment diagram is the graphical representation of the semantic model using a component layout language.

Having all this detailed information for the composition process, however, would cause the user of these fragments—assuming it is not the developer himself—a major distraction and it would provide way more information than necessary. Besides, the composition diagrams—the visual representation of a composition program (see later)—would get too large which is one thing we want to avoid in our tooling. Hence, an abstraction of the fragment developer viewpoint has been implemented. This additional view is called the fragment user viewpoint. Here, only the essential parts of the fragments are visible. The model itself is reduced to a simple named icon with small circles attached to it which symbolize the ports. An example for this viewpoint is depicted in Figure 2.8(c). The dashed circles symbolize configuring ports, empty circles are used for receiving ports and the black circles are contributing ports. Composition diagrams can be modeled in a language which is independent of the component description language. Reuseware provides a generic composition description language with a graphical syntax defined in GMF. Therefore, GMF is the composition layout language.

An actual composition program in Reuseware consists of at least two fragments and some ports. The composition is realized by composition steps with composition links. These links can either be configuration links, i.e. between two configuring ports4, or contribution links between a contributing and a receiving port. Contribution links also have arrows to show which one is the contributing and which one is the receiving

---

4 Remember that this alone does not make it a real composition
2.3. REUSEWARE COMPOSITION FRAMEWORK

Executing a composition program will result in the merging of the involved fragments and the output will be a bigger model. To be exact, not the fragments themselves are merged. Actually, the chosen components are copied and merged while the original fragments remain untouched. The composition is executed stepwise whereby the order in which the steps are executed is determined automatically. In Figure 2.9 the importance of such an order is shown. For the first composition, that is step 1, target is the target model but since it is possible to have more steps in a composition program this target can be the source of another composition step—that is step 2. It is obvious that we cannot execute both steps in parallel since you do not know beforehand what changes, i.e. configurations and additions, are applied to the model. Hence, step 1 has to be completed before step 2 can start. For a more detailed explanation please read [13].

In Figure 2.9 the fragments are labeled target or source. These terms are synonyms for the receiving and the contributing fragment.

To sum it up there are basically three things to do in order to model a valid composition.

1. **Choose target model:** That will be the receiving fragment you want to add functionality to. We will see later, that a target model does not necessarily have some content.

2. **Choose source model(s):** This is what we want to add or how we want to change the target model.

3. **Connect ports:** Connect the contributing and the receiving port and optionally two configuring ports.

Until now, we only discussed how the basics of a composition work, but there is one essential thing missing in Reuseware. The target model’s layout of a Reuseware composition was hardly what a user might hope for. Fragment developers start to model the fragments somewhere on the editor canvas. The problem is that fragment users have their own editor and they create their own model using only icons of the fragments. We assume that they intentionally place these icons in certain areas and that they hope for the actual fragments to appear right there in the target diagram. By using GMF we would have access to this layout information (as depicted in Figure 2.6) but unfortunately, Reuseware completely discards it which leaves the user totally disoriented. Additionally, the fragment diagrams partially overlap and, even worse,
if you want to use the same fragment several times—which, in some cases, you are allowed to in Reuseware—these overlap completely.

In this thesis we aim at tackling that very problem. We want Reuseware to generate user friendly diagrams no matter how often a fragment is reused and no matter where the user placed it. In order to create something in a user friendly way, however, it is useful to be certain about what that term really means in this environment. Therefore, it is necessary to get familiar with the Mental Map which is explained in the following chapter. Additionally, we present how we intend to enhance the layout creation process to meet the requirements of a design that is easy to understand.
Chapter 3

Layout Concepts

After introducing the basic technologies we require an explanation of two terms that will pop up every now and then throughout this thesis. The following concepts are very important in order to understand the relevance of this work since they are the basis for a user friendly software product using the previously introduced technologies. In this chapter, the terms Mental Map and Layout Adjustment will be explained. Firstly, in Section 3.1 the Mental Map—a psychological term which is used rather seldom in a software development environment—is introduced followed by the Layout Adjustment in Section 3.2 which should be more common to a software engineer.

3.1 Mental Map

Creating a layout for a diagram automatically is not as easy as one might assume. Sure, as a software engineer you are free to implement algorithms you think are either appropriate, nice looking regarding your sense of style or easy to realize. But what about the acceptance of your software? A user might have a completely different understanding of what is easy to read and he might prefer another algorithm or even create a layout himself. There is a great variety of algorithms to create layouts automatically and any one of them has its advantages and disadvantages. Implementing all of them would take a lot of effort.

Hence, in order to be accepted by most of the users, a tool should provide the possibility to develop a custom layout. This concept is called interactive graph visualization and it ensures a better acceptance than any predefined algorithm. In this field, the Mental Map plays a very important role. While laying out the different nodes and edges in a visual editor, the user subconsciously memorizes their location. Piece by piece a net is built in the user’s head that is not very different from a road map. In a model-driven software development process this road map is a fundamental concept to preserve the user’s orientation. When two (or more) models are composed, as this is the case for the Reuseware Composition Framework, it is like having two (or more) different road maps of the same town. Each of these road maps is displaying different streets and we just need to put them together, i.e. compose them.

There are some restrictions, though. We cannot put the northern part—or component—of that map to the south or twist and turn the different fragments in a way
CHAPTER 3. LAYOUT CONCEPTS

Figure 3.1: Fragment Integrity

the streets are not connected anymore. The same restrictions are applicable for a diagram composition as well. Users who laid out their street map, i.e. model, want it to look at least very close if not exactly like that after pressing the “compose” button. Some aspects of their Mental Map, e.g. the location of the nodes or the neighborhood relationship of two nodes, must not be touched throughout the composition process. Figure 3.1 depicts the aspect of the neighborhood relationship. Reused fragments need to be recognized as one piece within the resulting diagram and their nodes should not be scattered. Due to all these difficulties it is not enough to generate a layout automatically. Since every user wants a custom layout which has to be easily readable there needs to be some layout adjustment as well.

3.2 Layout Adjustment

With the knowledge about what the concept of the Mental Map of a user is we now need to make up our minds how to realize its preservation in Reuseware. The goals we want to achieve in order to preserve the user’s Mental Map are:

**Goal 1:** disjointness of nodes

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

**Goal 3:** compact design

In Section 2.3 we already gave a brief introduction about the Reuseware Composition Framework’s benefits and what it is still lacking. Until now there was no explicit layout algorithm implemented which led to diagrams like the one illustrated in Figure 3.2.

This is not what we wanted to achieve although it is a very compact design and therefore already meets our third goal. Compactness, however, is less important than
3.2. LAYOUT ADJUSTMENT

the disjointness of the nodes since the user has to be able to see every fragment included in the composed diagram. With the Mental Map concept setting the rules for a good design we have to think about solutions which follow them. On the one hand we have the automatic layout creation and a big variety of algorithms we just have to implement. On the other hand, as we will see soon, this might not be enough.

Let us take the algorithm for planarity [18] which is depicted in Figure 3.3 for an example. As a mathematician you might be amazed by how these simple calculation can make such a big difference. As a software engineer you might see it as a challenging task to implement this and you are happy someone else already made up their mind about algorithms which spares you that part. Customers, however, have their own sense of style and efficiency.

When we now start to compare the result of the planarity algorithm to the goals we wanted to achieve we recognize some dissonances. Firstly, the planarity algorithm does not really count for an example because it does not care for the disjointness of nodes but its purpose is to remove crossings of edges. It is, however, an algorithm for automatic layout creation and in Figure 3.3 there are no overlapping nodes in the result which is goal number one. In order to remove crossings of edges nodes have to be moved as this is the case for the C node in Figure 3.3. This clearly violates the second goal, i.e. that preserving the Mental Map means keeping the neighborhood relationship of the nodes. Although, for this simple example, this does not make a big difference, one might imagine users get confused when they have a bigger diagram and several of these changes.
Having an architecture like the one the Reuseware Composition Framework offers, there are some things that need to be realized to preserve the Mental Map. As mentioned in Section 2.3 there are two different levels of abstraction. One lower level—the fragment developer viewpoint—featuring the actual fragment models and one higher level—the fragment user viewpoint—for their reuse. Each of these levels has an own editor. The composition diagram editor has icons, i.e. abstractions, representing the actual fragment diagrams. These icons do not have the size of the fragments and although the user arranged them in a way that there are no overlaps, the composed diagram may have some. In order to preserve the Mental Map we have to merge the layout information from the two levels so that we map the information from the abstraction to the level with the real models.

By now it should be clear that with automatic layout creation alone we will not get very far. For that reason an additional step called *layout adjustment* has to be added to the modeling process. As we will see later it is not sufficient for some algorithms to execute that step once. In several cases there were still overlapping fragments left even after some iterations. Figure 3.4 shows the order for a modeling process in Reuseware. As a start, the composition model is developed by the fragment user by arranging the fragment abstractions using the composition editor. After that, the layout for the composed model is created automatically using the information taken from that editor and mapping it to the component description language. Once this is done, a check for overlaps is executed and in case there are such, the layout is adjusted.

Figure 3.4: Adjustment Step
Chapter 4

Layout Algorithms

This chapter is about the third step of Figure 3.4, i.e. the layout adjustment. This means that modeling and layout creation are done and we now focus on detecting overlaps and their removal.

Two of a great variety of algorithms for layout adjustment are introduced. In Section 4.1 we discuss the Horizontal Sorting and in Section 4.2, Uniform Scaling is presented. Advantages and disadvantages are described as well as problems that may occur with regard to the layout concepts discussed in the last chapter. You may wonder why we chose these two algorithms. Actually, there are several reasons. As a start, each of them is based on a different approach and forced us to adapt the tooling. This adjustment has to be made whenever there is another approach with different requirements to be implemented. Furthermore, Horizontal Sorting and Uniform Scaling are quite easy to implement and, thus, it is easy to explain their advantages and drawbacks.

4.1 Horizontal Sorting

The first algorithm for layout adjustment in the field of interactive graph visualization we found is called Horizontal Sorting [16]. As stated in the introduction, this is a rather simple algorithm because it basically does what its name implies. Starting on the left hand side of the diagram overlapping fragments are moved in x direction until they do not overlap with the previous fragment anymore. Although it sounds easy, there are still some things to do to achieve a correct sorting.

To achieve a design that goes by the rules of Horizontal Sorting a certain order of layout steps has to be followed.

1. Choose the next fragment
2. Check for overlaps
3. Arrange fragment

For the first fragment of the whole composition, there will not be any overlaps. Therefore, it is simply merged with the target diagram. In that case the location for
the fragment is taken completely from the information given by the user’s arranging in the composition diagram. From the second fragment on, things get a little more difficult though. However, if no overlaps are found, even now the fragments are moved to the location specified by the user. What will never change, no matter how many fragments are involved in the composition, are their y values.

One more aspect we need to have a closer look at is the definition of the first fragment. It showed that the order of the fragments for the composition is not arbitrary. Actually, that order must not be arbitrary. It is essential to begin with the one that has the lowest x value of all the fragments for a composition.

Figure 4.1: Importance of the order of composition steps

Figure 4.1 depicts the difference between the wrong (upper illustration) and the right choice (lower illustration). In that figure a merge of the composition diagram and the corresponding fragment diagrams is shown in a simplified way. The linked rectangles in the foreground are from the composition diagram and each of them represents a fragment diagram which is displayed in the background. The numerals within the rectangles symbolize the order for the composition. In the first case, the diagram with the higher x value is composed first. There is no overlap since the algorithm only sees one fragment at a time. Here, as stated above, the diagram is simply merged with the target diagram. Once the second diagram comes into play, however, an overlap will be detected and a layout adjustment has to be performed. Hence, the second diagram is moved in x direction. This adjustment would leave unused space on the left hand side of the target diagram. Since one of our goals was to create a compact diagram (Section 3.2, Goal 3), this variant would be inappropriate. Even worse, by moving the left neighbor of a diagram in a way that, after the composition, it is the right neighbor would completely destroy the user’s Mental Map. That is, because Goal 2 in Section 3.2 tells us to keep the neighborhood relationship if we want to create a user friendly layout. The second case, however, illustrates the correct
4.2. UNIFORM SCALING

choice. If there are overlaps they will be eliminated by moving the second diagram to the right. Declaring an order for the composition makes this algorithm deterministic.

In order to evaluate Horizontal Sorting regarding the goals set by the Mental Map we will create an imaginary example. Assume we have a rather big composition program with a lot of fragments to reuse. Software for managing big heterogeneous communication nets, for example, has to deal with a lot of different incoming data and, thus, has to be developed using many fragments. As a software engineer who does not know about the actual size of the diagram of a fragment since we only see an abstraction we create a composition diagram in a naive way. Since, with Reuseware, we want to provide means that offer a good overview, the abstractions will in almost every case be smaller than their corresponding fragment diagram. Hence, we most likely will design a small composition diagram not leaving too much space between the fragments in either direction, i.e. \( x \) and \( y \). This leads to a lot of overlaps and therefore, to a lot of work for our layout adjustment algorithm.

![Figure 4.2: Drawback of Horizontal Sorting](image)

Figure 4.2: Drawback of Horizontal Sorting

Sorting horizontally means that with each overlap the target diagram gets a little wider and the probability that the next fragment overlaps increases. What, in some cases, would require just a little correction in \( y \) direction, as depicted in Figure 4.2, will result in a big rearrangement in \( x \) direction. Option (b) in that figure represents the more clever rearrangement. It would keep the neighborhood relation of the fragments, remove the overlap and yield a compact diagram for result. Instead, our algorithm does what is depicted in option (a). This causes a violation of Goal 3, the compact design.

4.2 Uniform Scaling

The second approach for layout adjustment we chose is called Uniform Scaling [2]. This algorithm is based on a mathematical equation and yields a completely different result compared to Horizontal Sorting.

The algorithm looks as follows: \((a+s*(x-a), b+s*(y-b))\)
As depicted in Figure 4.3, the point \((a, b)\) is a center point and \((x, y)\) is the location of the fragment that needs to be moved. The factor \(s\) is a scale factor used to enlarge the amount the fragment is moved by. For automatization purposes, \((a, b)\) should not be chosen by the user. Instead, it should be computed. Since we know the location of each fragment from the composition diagram we are able to get the exact center point by simply taking the point \(P\left(\frac{c_2}{2}, \frac{d_2}{2}\right)\). \(c\) stands for the width of the composed model without adjustment and \(d\) is its height. Therefore, the whole composition diagram has to be visible for our algorithm. This, however, is a big difference to Horizontal Sorting. As you might remember, only one fragment at a time is needed for that sorting. Due to that difference, another problem occurred. Even though we do not need the fragments ordered for the algorithm to work the way we want it to, the ordering cannot be omitted. It is essential for the check for overlaps. Given a list of the involved fragments, we need to test each single component against all the other fragments in that list. In case of an overlap all fragments have to be moved. Unfortunately, there is no other way but to do it exactly like this. Imagine we would compute the new diagram with the above shown equation whenever there is a new fragment. In every step a new center point is needed and with every application of the equation the fragments are moved a bit away from each other. In that case the result would neither be compact nor would it obey the rule of neighborhood relationship and, thus, destroy the Mental Map.

At the time we decided on Uniform Scaling the usage of the information from the composition program was not thought through yet. There was a little hack which made Reuseware take over the fragment’s location from the fragment developer editor. If you were lucky, the fragment developer placed the diagrams at different locations. Otherwise, the scaling equation would have no effect on the result. However, even if the developer did spread the diagrams, the second reuse of the exact same fragment would cause a perfect overlap between them and, thus, Uniform Scaling would move both to the same location. For that, we first thought of an algorithm to move one of the fragments a little bit so that the layout adjustment has a chance to deliver a good result. The attempts were not very successful and, besides, without the arranging according to the information from the composition diagram, the user’s Mental Map would not be supported.

Luckily, we were given the required information and we were able to adapt Uni-
form Scaling to our needs. By that time we thought it would be the perfect algorithm for us. Some tests with small examples delivered satisfying results. When we tested some more, however, we had to admit that there are still some disadvantages to this algorithm. First of all, it is most likely that—depending on the scale factor $s$—after applying the algorithm only once the result still has overlaps. This forced us to do an overlap check again and after that execute the algorithm once more. Horizontal Sorting, as you will remember, did not require such a recursion. Along with this recursion the effect of the next two problems get even worse.

![Figure 4.4: Unused space due to Uniform Scaling](image)

The first of these problems is the fact, that you cannot rearrange parts of the diagram. Checking the whole diagram for overlaps and adjust all its parts when there are even only two fragments that hardly touch each other yields a diagram like the one depicted in Figure 4.4. With each overlap and, hence, with each layout adjustment the unused space grows in diameter. This result would not obey to our goal to create a compact design.

![Figure 4.5: Splitting off of fragment diagrams](image)

The second problem, that might occur also forces the user to rearrange the result in order to adjust it. Assuming the user placed a fragment somewhat away from the main group in his composition diagram as it is shown in Figure 4.5. The center
point of that diagram would be somewhere between the main group and that extra fragment which caused this fragment to get split off even more. Depending on the size of the screen the user is looking at the fragment will sooner or later get out of sight and has to be dragged back in manually.

As you can see, the Uniform Scaling is not a perfect algorithm either. Nevertheless, it appropriately demonstrates a completely different approach compared to Horizontal Sorting and what has to be done to adapt it for layout adjustment in Reuseware. Some vital requirements for the Horizontal Sorting are unnecessary when using Uniform Scaling and vice versa. Maybe other more elaborated approaches require even a more sophisticated adaption.
Chapter 5

Implementation

After the presentation of the requirements in the last chapters we now have a look at how they are implemented. Due to the fact that Reuseware was totally lacking that layout adjustment part, some extensions had to be made. In Section 5.1 we give a brief overview of these extensions and an explanation of the concepts. Right after that, that is in Section 5.2, the class CompositionDiagramUtil which plays a big role in this new design will be introduced. In Section 5.3 there will be more detailed information regarding the extensions of Reuseware including code snippets.

5.1 Reuseware extensions overview

The Reuseware Composition Framework, as it was at the time this thesis was started was not appropriately designed to manage the task at hand. Reuseware was designed to reuse fragments that are graphically represented using GMF or TOPCASED for component layout languages. Additionally, it was possible to implement one (and only one) algorithm for layout adjustment. Therefore, before we can implement the algorithms from Chapter 4 to adjust the layout of diagrams, we should spend at least one or two thoughts on how to adjust Reuseware’s design. Software engineers know that decisions on a certain design never are easy ones. There are a few things you need to consider like whether your design is of any help for future tasks or if that design is simple or if its complexity can be reduced in some way.

The first decision was to create different plugins for GMF and TOPCASED. Since they are the only two component layout languages we used and due to the fact that there are some language dependent parts in our solution having them separated like that makes it easier to understand the code. Otherwise, we would have had to mix the code for both these languages with a lot of if-statements. This was the case for Reuseware before this thesis.

Figure 5.1 shows how the Reuseware framework was extended and why we decided for this architecture becomes quite obvious if you know about the purpose of each of the components. Only the Merger and the Providers are language dependent and, thus, need to be implemented both in the TOPCASED plugin and in the GMF plugin.
5.1.1 Diagram Composer

In order to compose fragments the user is allowed to define a Composer. This extension point gives the opportunity to combine the extensions shown on the right hand side of Figure 5.1. The Providers are required for every composition which is why they do not need to be specified in the composer.

Listing 5.1: Diagram Composer

```
<composer
  id="..."
  comparator="..."
  arranger="..."
  merger="...">
  ...
</composer>
```

Listing 5.1 shows such a composer definition. While comparator and arranger can be left out, id and merger are required attributes. The id needs to be unique since it is for choosing the Layout Composition Strategy (cf. Chapter 6). Furthermore, without naming the merger there will not be a composition of fragments (cf. Sections 5.1.5 and 5.3.5). Comparators and arrangers are only required if the user wants the layout adjusted. Until now there are two arrangers and one comparator to choose from but we will see later that we have to consider implementing alternatives in order to provide a more sophisticated solution.
5.1. REUSEWARE EXTENSIONS OVERVIEW

5.1.2 DiagramInformationProvider

When we introduced the Reuseware Composition Framework in Section 2.3 we already mentioned that we are dealing with two different levels of software engineering. On the one hand there is the fragment developer who provides reusable components defined in a component description language. On the other hand a fragment user works on a higher level of abstraction and simply uses the previously created fragments, composing them to create a custom software product.

Layout adjustment, however, requires some kind of a merge of the two levels since, for example, in the composition editor we just have an abstraction of the fragments and no information about their actual width and height. What we have, though, is the information how the user wants the fragments arranged in the composed diagram. SourceDiagramInformationProvider (SDIProvider) and TargetDiagramInformationProvider (TDIProvider) are, therefore, two essential extensions to the framework providing data about the source, i.e. the fragment meant for reuse, and the target, i.e. the diagram developed by the fragment user. Sources and targets, in this case, must not be confused with the contributing and receiving diagrams introduced earlier when they were input and outcome of the composition process in Reuseware. We have to keep in mind that a fragment component always has two representations. One fragment diagram created using a component layout language (source) and an icon on the more abstract level (target). This means that even receiving diagrams have source diagram information as well as contributing diagrams have target diagram information.

We faced some problems regarding unwanted and even invisible nodes within the diagrams which yielded wrong values for location, width or height. How those problems were tackled will be explained in Section 5.3.1.

5.1.3 Comparator

A few tests later and after double-checking the correctness of the written code we wondered why this is not working the way it should. The results for the composition using one specific algorithm differed slightly each time the program was executed. By running even more tests and concentrating on the differences we discovered that, in some cases, one fragment diagram was displayed in the foreground and in other cases it was displayed in the background, i.e. it was overlapped by another fragment diagram. Luckily, at that time the algorithm was not as sophisticated as to remove all overlaps. This difference in the display of a single fragment, however, led to the conclusion that it was not the implementation of the algorithm that delivered the wrong results but that it had to do with the preconditions.

It turned out that Reuseware had an indeterministic way of choosing the order of the input elements for a composition. There are some algorithms, though, that need a certain order to work correctly. Therefore, the opportunity to determine such an order is given by the Comparator extension point. Adapting its compare functionality is an easy way to implement other algorithms with different requirements in the future. In Section 4.1 we discussed why sorting the elements for a composition is inevitable for the algorithm Horizontal Sorting. Furthermore, in Section 4.2 it was
explained how the order serves for finding overlaps as a first step to Uniform Scaling.

5.1.4 Arranger

When the preparation is completed, the actual arrangement, i.e. writing the code for the adjustment algorithm, can be done. The complexity of this task depends on which one out of the variety of algorithms we select. Checking for overlaps is supposed to be easy but as we will see it needs to be adapted to the adjustment algorithm. Some overlaps just cannot occur and, therefore, checking for them is redundant and can be omitted. Each Arranger represents an adjustment algorithm and since all required data is independent of the involved composition and component description layout languages, the arrangers themselves are independent of those.

Since we decided to implement Horizontal Sorting and Uniform Scaling we needed two different kinds of Arranger. Namely, those kinds are SingleSourceDiagramArranger which is used for Horizontal Sorting and MultiSourceDiagramArranger which is the one that is necessary for Uniform Scaling. The reason for this difference will be explained in further detail in Section 5.3.4.

5.1.5 Merger

Having done all this we can imagine our new diagram being a puzzle with its parts in the right place but the whole thing is kind of floating an inch above the table and the parts are not connected. This physical merge of the diagrams that were input to the composition to an actual result is realized by a Merger. This task, however, is not an easy one as well because, as we will see, there may be some nodes of a fragment diagram that you do not want in the result. Hence, there is a need to exclude those elements. It is a similar problem to the one with the DiagramInformationProviders (cf. Section 5.3.5).

5.2 The class CompositionDiagramUtil

Once we have such a big band we need a good conductor who is responsible for the individuals to harmonize. In this solution the conductor is called CompositionDiagramUtil. With the rough overview given in the last section we can now have a closer look at and understand how the extensions applied to Reuseware are supposed to work together.
As a start, we have to have a look at two basic classes which will be used in the following sections. These classes are `FragmentDiagram` and `Bounds`. `Bounds`, as shown in Listing 5.2, contains the get and set methods for the layout information. Its constructors define default values which, on the one hand, is necessary but, on the other hand, led to some confusing composition results as we will discuss later.

```
public class Bounds {
    protected int x;
    protected int y;
    protected int width;
    protected int height;

    public Bounds();
    public Bounds(int x, int y);
    public Bounds(int x, int y, int width, int height);

    public int getX();
    public void setX(int x);
    public int getY();
    public void setY(int y);
    public int getWidth();
    public void setWidth(int width);
    public int getHeight();
    public void setHeight(int height);
}
```

Listing 5.2: Bounds
CHAPTER 5. IMPLEMENTATION

The class `FragmentDiagram` shown in Listing 5.3 contains the methods for gathering the specific `Bounds` of a diagram. A diagram—as we already discussed in Section 5.1.2—has two different kinds of bounds. Therefore, we need `getSourceBounds()` to get and set the source bounds and `getTargetBounds()` for the target bounds.

Having two different layout adjustment algorithms whereof each one requires different handling the orchestration by `CompositionDiagramUtil` is essential. Not only is it used to register all the parts needed for a composition, i.e. InformationProvider, Comparator, Arranger and Merger, but it is also responsible to select the right approach for the two different forms of Arrangers.

```
// get target bounds of receiving diagram
for (SourceDiagramInformationProvider provider : sdiProviders)
{
    if (provider.canProvide(receivingDiagram, receivingContext))
    {
        provider.provideBounds(receivingDiagram, receivingContext);
        break;
    }
}
```

Listing 5.4: Providing of source diagram information
5.2. THE CLASS COMPOSITIONDIAGRAMUTIL

There is a certain order required regarding the steps of a composition for the algorithms to work properly. First of all, the diagram information for the contributing as well as for the receiving diagram has to be gathered. Without that information, no ordering of the fragments would be possible. In case the fragments are not ordered, overlap checks would most likely yield wrong results and hence, the complete layout adjustment would fail. Listing 5.4 shows how that is done for the source diagram information for a receivingDiagram. The implementation of canProvide is for determining whether the given FragmentDiagram is of the layout language the provider supports, while provideBounds sets the actual values for this diagram.

```
// sort the list of contributing diagrams for composition order
DiagramComparator comparator = composer.getComparator();
if (comparator != null) {
    Collections.sort(contributingDiagrams, comparator);
}
```

Listing 5.5: Comparison of contributing fragments

The second task CompositionDiagramUtil performs is the initialization of the comparator. Comparing and putting the contributing diagrams in order is inevitable if we want our algorithms to work as expected. What happens if this step is skipped is described in Chapter 4. Neither of the algorithms we use would work properly. In Listing 5.5 this initialization is shown.

When the layout information was gathered and the fragments are in the correct order for a composition their arrangement according to the rules discussed in Chapter 3 can take place. As stated above, due to the different approaches for layout adjustment, we had to implement two kinds of Arrangers. For MultiSourceDiagramArranger, the handling is quite easy. As you will see later in the more detailed description of the UniformScalingArranger—which implements MultiSourceDiagramArranger—the lines of code shown in Listing 5.6 is enough to get a correct result for the algorithm Uniform Scaling.

```
((MultiSourceDiagramArranger)arranger).arrange(
    contributingDiagrams, receivingDiagram);
composer.getMerger().merge(contributingDiagrams, receivingDiagram, receivingContents);
```

Listing 5.6: Arranging for MSDArranger

It looks that simple because all the functionality required to perform the algorithm is within the class UniformScalingArranger. Unlike that, the second approach requires some more steps. In Section 4.1 we explained that, for the Horizontal Sorting, only one fragment at a time is in focus. Hence, to perform a whole composition, the fragments have to be merged one by one. When we merge a contributing with a receiving diagram, however, the layout information, that is the source diagram information, of the receiving diagram changes. Therefore, we have to update our layout information of the receiving diagram. That means that, for SingleSourceDiagramArrangers, there are three steps to be executed. These steps would be to arrange, to merge and
to provide source diagram information.

```java
// arrange
((SingleSourceDiagramArranger) arranger).arrange( contributingDiagram, receivingDiagram);

// merge
composer.getMerger().merge(Collections.singletonList(contributingDiagram), receivingDiagram,
                             receivingContents);

// re-compute source bounds of the receiving diagram that is now merged with the contributing diagram
for (SourceDiagramInformationProvider provider : sdiProviders)
{
    if (provider.canProvide(receivingDiagram))
    {
        provider.provideBounds(receivingDiagram);
        break;
    }
}
```

Listing 5.7: Arranging for SSDArranger

Listing 5.7 contains the source code for these three steps.

To sum it up, CompositionDiagramUtil is responsible for calling the right operations in the correct order. When it comes to arranging the fragment diagrams with respect to the rules of the two different layout adjustment algorithms, this class makes a distinction that is necessary for the approaches to work correctly.

5.3 Reuseware extensions in detail

After we had the new plugins introduced and roughly explained their roles in our approach there is a more detailed insight provided in the next sections. In Section 5.3.1 and Section 5.3.2—under the headline of DiagramInformationProvider—there is an introduction of the SourceDiagramInformationProvider and the TargetDiagramInformationProvider. Right after that, that is in Section 5.3.3, the functionality of the Comparator—the XYPositionComparator—is explained. Since we had to implement only one sorting for both our algorithms, there was only one Comparator needed. In Section 5.3.4 we have a closer look at the Arrangers. As we will see, the actual arranging was not the hard part but finding the overlaps was. The last section, i.e. Section 5.3.5, is about the Mergers.

5.3.1 SourceDiagramInformationProvider

One goal of the new extensions implemented in Reuseware was to realize the layout adjustment as language independent as possible. Since both, the composition diagram editor and the fragment diagram editor, may be based on different layout languages (e.g. GMF, TOPCASED) this abstraction seemed to be quite useful for a reuse of the implementations. SourceDiagramInformationProvider is an interface with two methods, canProvide(FragmentDiagram fragmentDiagram) and
5.3. REUSEWARE EXTENSIONS IN DETAIL

provideBounds(FragmentDiagram fragmentDiagram). Classes that implement this interface are used to compute the bounding box (i.e., x, y, width and height) of the source diagrams that were modeled in the fragment diagram editor. Hence, these classes have to be layout language specific. Listing 5.8 shows the computation of the biggest x value of a GMF diagram. This value is then needed to compute the width of the diagram.

```java
private static int findMaxX(EObject diagram){
    TreeIterator<EObject> diag = diagram.eAllContents();
    int max = 0;
    while(diag.hasNext()){;
        EObject nodes = diag.next();
        EObject node = findSemanticModelElement(nodes);
        if(node==null && nodes instanceof Node){
            Node nodeEl = (Node) nodes;
            if(nodeEl.getType().equals("Note")){
                LayoutConstraint location = ((Node) nodeEl).
                getLayoutConstraint();
                LayoutConstraint size = ((Node) nodeEl).
                getLayoutConstraint();
                if(location instanceof Location && size instanceof Size){
                    if(((Location) location).getX()+(Size) size).getWidth()>max){
                        max = ((Location) location).getX()+(Size) size).getWidth();
                    }
                }else{
                    if(node!=null && nodes instanceof Node){
                        Node nodeEl = (Node) nodes;
                        LayoutConstraint location = ((Node) nodeEl).
                        getLayoutConstraint();
                        LayoutConstraint size = ((Node) nodeEl).
                        getLayoutConstraint();
                        if(location instanceof Location && size instanceof Size){
                            if(((Location) location).getX()+(Size) size).getWidth()>max){
                                max = ((Location) location).getX()+(Size) size).getWidth();
                            }
                        }
                    }
                }
        }
    }
}
```

1This is realized using the GMFSourceDiagramInformationProvider. These implementations, however, depend on the layout language. Therefore, an equivalent implementation is required for fragment editors not based on GMF.
Listing 5.8: Computation of the biggest x value

As a start a TreeIterator has to be defined since a diagram has a tree-like structure and the biggest x value can be found not only in the root node but also in one of the children or even in a leaf. After running a few tests it turned out the computation was error-prone. According to the delivered values, some diagrams were wider and higher than they appeared in the composed diagrams.

Figure 5.2: Nodes containing additional information

Figure 5.2 depicts the reason for this problem. Additional information for the composition was added by the fragment developer when he modeled the components. Since this information was contained in a Text node, the algorithm was applied to these nodes as well which resulted in unexpected and even unwanted return values. By implementing the method findSemanticModelElement(nodes) we check whether a node is a valid part of the fragment model or not. If it is not, its values are discarded during the computation of the bounding box. There is, however, a type of node that needs to be considered although it is not part of the model. In Figure 5.2 this kind of node is displayed as a green rectangle labeled “DHCP Client” in the background. As you will see in Chapter 6 this is a specialty to our CIM examples. However, you might find similar thing in other examples and hence, this is a good opportunity to show how it can be handled.

Listing 5.9: Check for nodes of type Note

Those nodes are of type Note and although the check for semantic model elements returns null it is the root node of almost all fragment diagrams of the CIM example. Thus, we had to check for that and Listing 5.9 shows the solution to that problem. The if-expression determines the type of the node and the bounds will be considered
5.3. REUSEWARE EXTENSIONS IN DETAIL

when it returns true. Since this is again a language dependent implementation, it has to be adapted to the notation model of the new layout language in case a different editor is used to model the fragments.

Setting the new values for width, height, x and y is now done by the FragmentDiagram's method getSourceBounds(). It returns the Bounds which can then be set by the according setter methods. If we chose not to set any of those values they will keep their defaults. Since the default values are -1 which could rise problems in the layout adjustment we decided to reset the defaults for certain nodes to reasonable values (height: 10px, width: 20px). When we use the GMF editor width and height of nodes are set to -1 if we just take the item from the tool bar without resizing it.

5.3.2 TargetDiagramInformationProvider

Interactive graph visualization is a concept which aims at involving the user's choices in the layout process as already explained in Chapter 3. To find out about how the user wants to arrange his diagrams a tool is needed which allows the user to lay out the model. Reuseware provides a composition diagram editor for this task in which icons represent the fragment diagrams. Users now have the opportunity to arrange the composition diagram in a way they think the resulting diagram is good to read. By default, this editor is based on GMF as layout language. When the user finished the layout process the diagram information is stored in a fcidi file which is an XMI file representing the GMF notation from the diagram. Although, as stated above, the default layout language for the editor is GMF, we need an adaptable implementation in case of a future change.

The TargetDiagramInformationProvider provides the Bounds of the composition diagram just as the SourceDiagramInformationProvider does it for the component diagrams. There is, however, a difference in the use of the provided information. For the implementation of the layout algorithms described in Chapter 4, the width and height of composition diagrams is not needed. What is essential for the algorithms and the user centered design is where the user placed the icons that represent the involved fragment diagrams, i.e. the exact location in terms of the x and y values, since this is necessary to preserve the Mental Map. When we started implementing the algorithms for layout adjustment we did not know which problems we would face. It appeared that each of them needed different handling and preparation before it worked the way it was supposed to. The Horizontal Sorting requires a sorting of the composition steps and only one source diagram at a time (cf. Section 4.1) while the Uniform Scaling works fine without a sorting of the composition steps but needs all the source diagrams for the computation of the center point (cf. Section 4.2).

Similar to the SourceDiagramInformationProvider, the default values for x, y, width and height are -1. When the Reuseware Composition Framework was introduced in Chapter 2.3 we already mentioned that the information from the composition diagram was completely discarded which resulted in a layout where the single source diagrams were stacked. Due to that, the results were impossible to read without

---

2 x, y, width and height
3 FragmentCompositionDiagramInformation
4 It does need a sorting to check for overlaps, though.
manual adjustment. However, this means that the default implementation of the TargetDiagramInformationProvider does not change anything. By gathering the layout information and providing it for the layout algorithms we are given the opportunity to create a user centered design.

5.3.3 Comparator

Comparators are responsible for specifying a certain order of composition. The necessity for such an order has been detected when the Horizontal Sorting was implemented. Given a set of contributing diagrams and a receiving diagram the algorithm produced a different result each time it was executed. Uniform Scaling, however, did not show this indeterminism. Hence, there was either an implementation error or the preconditions were not fulfilled. It turned out that it was the latter. The Reuseware Composition Framework had an indeterministic algorithm to choose the next contributing diagram for the composition. To fix this problem, the Comparators had to be integrated. Although Uniform Scaling works fine without sorting, it’s overlap check does not. The reason we did not notice that in the first place is, that the algorithm yields reasonable results for simple examples even without the sorting. Additionally, the overlap check was not as sophisticated at that time as it is now. However, this means, that neither Horizontal Sorting nor Uniform Scaling generate valid results without Comparators providing a sorting. Due to the fact that they work with the information provided by the TargetDiagramInformationProvider, Comparators are independent of the composition layout language as well as of the component layout language.

```java
// starting with lowest x value
if (fd1.getTargetBounds().getX() < fd2.getTargetBounds().getX()) {
    return -1;
} else {
    // if fd1.x = fd2.x the lowest y value wins
    if ((fd1.getTargetBounds().getX()==fd2.getTargetBounds().getX())
        &&(fd1.getTargetBounds().getY()<fd2.getTargetBounds().getY())){
        return -1;
    }
    return 1;
}
```

Listing 5.10: Comparison of two FragmentDiagrams

The XYPositionComparator compares two FragmentDiagrams. More precisely, their locations in the composition diagram are compared. Out of all the fragments involved in the composition the one with the lowest x value will be the first. If the user moved the icons so that they have the same x value, the one with the lowest y value of those fragment diagrams will be the next for the composition. Listing 5.10 shows the implementation of the method compare(FragmentDiagram fd1, FragmentDiagram fd2). In our framework, we require an interface called DiagramComparator which is implemented by the XYPositionComparator. This interface extends the class java.util.Comparator where the compare method is defined.
5.3.4 Arranger

The Arranger performs the actual task of rearranging the given fragments according to the algorithm of choice. A major advantage of the new architecture is the language independence of the arrangers. This allows for easy adaptation of our solution to other layout languages in the future.

HorizontalArranger

The HorizontalArranger realizes the Horizontal Sorting which is described in Section 4.1. Given the next contributing diagram—the order has been specified in advance by the XYPositionComparator—and the receiving diagram this arranger checks for overlaps and sets the new coordinates for the contributing diagram. These coordinates can be taken either from the composition diagram, i.e. the TargetDiagramInformationProvider, or they are computed according to the rules of Horizontal Sorting which depends on the result of the overlap test.

```java
private static boolean isOverlap (FragmentDiagram contributingDiagram, FragmentDiagram receivingDiagram)
{
    int targety = receivingDiagram.getSourceBounds().getY();
    int tawidth = receivingDiagram.getSourceBounds().getWidth() +
                   receivingDiagram.getSourceBounds().getX();
    int theight= receivingDiagram.getSourceBounds().getHeight();
    int sourcex = contributingDiagram.getTargetBounds().getX();
    int sourcey = contributingDiagram.getTargetBounds().getY();
    int soheight= contributingDiagram.getSourceBounds().getHeight();

    boolean overlap = false;

    //S(sourcex,sourcey) within the bounding box of the target
    if(sourcex<tawidth &&
       sourcey>targety &&
       sourcey<(targety+theight)){
        overlap = true;
    }
    //S(sourcex,(sourcey+soheight)) within the bounding box of
    //the target
    else{
        if(sourcex<tawidth &&
           (sourcey+soheight)>targety &&
           (sourcey+soheight)<(targety+theight)){
            overlap = true;
        }
    }
    //S(sourcex,receivingDiagramy), S(sourcex,(sourcey+soheight))
    //NOT within the bounds but (sourcex > targetx) and
    //((sourcex < (targetx+tawidth))
    else{
        if(sourcex<tawidth &&
           sourcey < targety &&
```
CHAPTER 5. IMPLEMENTATION

Listing 5.11: isOverlap(FragmentDiagram, FragmentDiagram) of HorizontalArranger

Listing 5.11 shows the method isOverlap(FragmentDiagram, FragmentDiagram) which returns true if there are overlaps and false in case there are none.

For the first step in the composition isOverlap() will always return false since the receiving diagram is empty\(^5\). In this case and whenever the return value is false the contributingDiagram is moved to the position specified in the composition diagram. That arranging is performed by the method sortHorizontally(FragmentDiagram, FragmentDiagram) which is shown in Listing 5.12.

```
private static void sortHorizontally(FragmentDiagram contributingDiagram, FragmentDiagram receivingDiagram)
{
    // if the target diagram is empty, set the x value of the
    // source according to the value given by the composition
    // diagram
    if (receivingDiagram.getSourceBounds().getWidth() < 0)
    {
        contributingDiagram.getTargetBounds().setX(
        contributingDiagram.getTargetBounds().getX());
    }
    else{
        contributingDiagram.getTargetBounds().setX(
        receivingDiagram.getSourceBounds().getWidth() +
        receivingDiagram.getSourceBounds().getX());
    }
    contributingDiagram.getTargetBounds().setY(
    contributingDiagram.getTargetBounds().getY());
}
```

Listing 5.12: The method sortHorizontally(FragmentDiagram, FragmentDiagram)

The actual sorting, i.e. the update of the location the diagram has to be moved to, is rather simple. Horizontal Sorting is all about the x value of a diagram and does not care about the y value. Hence, this value can be taken from the composition diagram which supports the user centered design this approach is aiming at. After each composition step the width of the receiving diagram is increased by the width of the last contributing diagram that has been merged with it.

\(^5\)What happens if the receiving diagram is not empty will be shown in Section 6.2
5.3. REUSEWARE EXTENSIONS IN DETAIL

UniformScalingArranger

For the Uniform Scaling another approach is required. As introduced in Section 4.2 the algorithm is based on an equation which moves all fragments away from a given center point. Since the composition diagrams differ in the amount of involved fragments and in size, this center point has to be computed for each of these diagrams. Furthermore, we have to consider that one execution of the adjustment algorithm might not be enough. In Section 4.2 we discussed that, in some cases, there needs to be a recursion because after the first step there are still overlaps left. In case of such a recursion one cannot take the previously computed center point since the location of the involved fragments has been changed. Thus, a re-computation of the center point is required.

```java
private static int findCenterX(List<FragmentDiagram> diagram)
{
    ListIterator<FragmentDiagram> fragit = diagram.listIterator();
    int x = 0;
    int x1 = 2000;
    while(fragit.hasNext()){
        FragmentDiagram diag = fragit.next();
        // find the biggest x value, starting at x=0
        if(((diag.getTargetBounds().getX()+diag.getSourceBounds().getWidth())>x){
            x = diag.getTargetBounds().getX()+diag.getSourceBounds().getWidth();
        }
        // usually the first fragment does not start at x=0
        // so you have to add the white space again for
        // a correct computation
        if(diag.getTargetBounds().getX()<x1){
            x1=diag.getTargetBounds().getX();
        }
    }
    return (x1+x)/2;
}
```

Listing 5.13: findCenterX(List<FragmentDiagram>)

Listing 5.13 shows how the x value of the center point is computed. Actually, the method findCenterX computes two different x values. The reason for this is that we require the biggest x value of all the involved fragments—which is x—as well as the lowest—x1. The latter is used to get the real center point because users usually do not place the first fragment at the location (0, 0). This means, that we have to add the width of the unused space to the maximum x value before we divide it by 2. Besides, what we just discussed has to be applied for the computation of the y value as well.
private static boolean isOverlap(List<FragmentDiagram> diagram) {
    ListIterator<FragmentDiagram> fragit = diagram.listIterator();
    boolean overlap = false;
    int actDiag = 0;

    // Fragments are already sorted by XYPositionComparator!!
    // from second contributing diagram (index=1) on
    // each one has to be compared to it's predecessors
    while (fragit.hasNext()) {
        actDiag++;
        FragmentDiagram source1 = fragit.next();
        for (int i = actDiag; i < diagram.size(); i++) {
            FragmentDiagram source2 = diagram.get(i);
            int source1x1 = source1.getTargetBounds().getX() + source1.getSourceBounds().getWidth();
            int source1y1 = source1.getTargetBounds().getY();
            int source1y2 = source1y1 + source1.getSourceBounds().getHeight();
            int source2x1 = source2.getTargetBounds().getX();
            int source2y1 = source2.getTargetBounds().getY();
            int source2y2 = source2y1 + source2.getSourceBounds().getHeight();

            // only two options for overlaps because the
            // others are eliminated by the sorting
            if ((source1x1 > source2x1) && (source1y2 > source2y1) &&
                (source2y2 > source1y1)) {
                overlap = true;
            } else {
                if ((source1x1 > source2x1) && (source1y1 > source2y1) &&
                    (source2y2 > source1y1)) {
                    overlap = true;
                }
            }
        }
    }
    return overlap;
}

Listing 5.14: isOverlap(List<FragmentDiagram>) of UniformScalingArranger

When we computed the center point we are all set to rearrange the layout. Prior
to every adjustment recursion an overlap check has to be performed. To find overlaps
in the Horizontal Sorting algorithm was not too difficult since we had one fragment at
a time and we only had to compare it to the bounding box of the receiving diagram.
This time, however, we need to compare each fragment of the list of fragments to all
it’s successors. It is possible that fragment one overlaps with fragment three but not
with fragment two. Assuming that the fragments two and three do not overlap either,
the adjustment algorithm would not be triggered if we only compared a fragment with one successor.

In Listing 5.14 the solution to that problem is presented. The for-statement iterates over all fragments in the list that have a higher index value than the fragment in focus. With the sorting of the fragments by XYPositionComparator prior to the check, some options for overlaps cannot occur anymore. The ones that are still possible are covered by the two if-statements.

In case isOverlap(List<FragmentDiagram>) returns true, the method scaleUnified(List<FragmentDiagram>, int, int) is executed, which applies the actual adjustment algorithm. Listing 5.15 provides an insight into that method.

```java
private static void scaleUnified(List<FragmentDiagram> diagram, int centerx, int centery){
    //((a+s*(x−a)),b+s*(y−b))
    int scaleFactor = 2;
    int oldx = 0;
    int oldy = 0;
    ListIterator<FragmentDiagram> fragit = diagram.listIterator();
    while(fragit.hasNext()){,
        FragmentDiagram diag = fragit.next();
        oldx = diag.getTargetBounds().getX();
        oldy = diag.getTargetBounds().getY();
        diag.getTargetBounds().setX(centerx+scaleFactor*(oldx−centerx));
        diag.getTargetBounds().setY(centery+scaleFactor*(oldy−centery));
    }
}
```

Listing 5.15: The method scaleUnified(List<FragmentDiagram>, int, int)

In Section 4.2 we discussed the importance of the scale factor $s$. It has to be chosen carefully since, if it is too big, the unused space in the center of the diagram—as depicted in Figure 4.4—rapidly grows in diameter. On the other hand, choosing a small factor would slow down the process because it would take more steps to achieve a disjointness of all nodes.

### 5.3.5 Merger

What has been done so far was gathering the layout information from both the fragment diagrams and the composition diagram, sorting the fragments according to their x values and arranging them using one of the two layout adjustment algorithms. As stated in Section 5.1.5 there is one last step missing for the composition process to be completed. We have to physically merge the contributing diagrams with the receiving diagram.
Due to the fact that this task is performed on the level of the fragment diagrams, Mergers are layout language specific. This means that, as well as for the SourceDiagramInformationProvider, we have to implement mergers for each component layout language. Listing 5.16 provides the code for the GMFMerger’s method mergeGMF(EObject, int, int).

```java
private static void mergeGMF(EObject contributor, int x, int y)
{
    Iterator<EObject> sdiag = contributor.eAllContents();
    int oldx = 0; int boundx = 0;
    int boundy = 0; int oldy = 0;
    boolean change = true;
    while(change){
        change = false;
        oldx = ((Location) location).getX();
        oldy = ((Location) location).getY();

        //new x value to avoid overlaps
        bounds.setX(x);
        bounds.setY(y);
        boundx = x;
        boundy = y;
    }
    while(sdiag.hasNext()){
        EObject followingNodes = sdiag.next();
        if(followingNodes instanceof Node){
           LayoutConstraint location = ((Node) followingNodes).getLayoutConstraint();
           LayoutConstraint size = ((Node) followingNodes).getLayoutConstraint();

            if(location instanceof Location && size instanceof Size){
                Bounds bounds = NotationFactory.eINSTANCE.createBounds();

                //keeps the original size of the node
                bounds.setHeight(((Size) size).getHeight());
                bounds.setWidth(((Size) size).getWidth());

                //sets the new coordinates
                bounds.setX(((Location) location).getX()+ (boundx-oldx));
                bounds.setY(((Location) location).getY()+ (boundy-oldy));
            }
        }
    }
}
```
5.3. REUSEWARE EXTENSIONS IN DETAIL

Given the location to which the fragment has to be moved in the composed diagram (int x, int y) we just need to execute this rearranging. Therefore, each node of the diagram has to be moved which is why the TreeIterator has to traverse the whole tree. Needless to say that we cannot move each node to the given point. Although this would keep the Mental Map, the fragment diagrams would be unreadable. Hence, only one of the nodes—the root node—is put to that location and the following ones are moved relatively to their original position. oldx and oldy together are the original location of the root node of the fragment diagram and boundx and boundy determine the new location. Both points are required for rearranging the following nodes as it is shown in the second while-statement. In order to keep height and width of the nodes we had to write it because otherwise, those would be set to their default values. Besides, the class Bounds, in this case, is not the one specified by us but it is part of the GMF notation metamodel (cf. Figure 2.5).

That there are differences between the given layout languages can be seen in the approach required for the mergeTopcased-method implemented in the TopcasedMerger (not detailed here). In comparison to GMF, we do not need to move all nodes. In fact, the nodes are connected and if the right node is moved, the rest is arranged automatically. However, finding this right node is the problem which needs to be solved. There are nodes located at (0, 0) that cannot be moved and we simply took the first one that can be.

After this presentation of the basic concepts, the technology and the implementation we want to show how our solution works for the layout languages we introduced. In the following chapter we will provide two examples step by step.
Chapter 6

Evaluation

Theoretically, the implementation of layout adjustment in Reuseware should help to improve the readability of the diagrams. That it really does will be shown in this chapter. In Section 6.1 a composition using GMF for component layout language will be presented. After that, Section 6.2 provides an example for TOPCASED.

6.1 GMF example

For this first example we use CIM [15] for the component description language and GMF is the layout language. One specialty of this example is its level structure. It is a special hierarchy in order to reuse previously defined models or reuse already composed models. On level 1 the models defined by the fragment developer are stored. These can be used by a composition program modeled on level 2. The results of those compositions can then be reused by a composition program on level 3 and so on.

![Figure 6.1: Representations of a fragment](image)
As mentioned, there are two layers of abstraction involved in the composition process. In Figure 6.1 the three graphical representations of one fragment are depicted. There is the possibility to define a special syntax for the representation of a fragment in the fragment user viewpoint. One can, for example, hide contributing and receiving ports and assign pictures to the fragments as it has been done in Figure 6.1(a). However, to show all the details of a composition, we decided not to use such a specific syntax. Figure 6.1(b) depicts the default shape for the BuiltInEthernetHub fragment which is the one we will use in this section.

The BuiltInEthernetHub is a fragment modeled by a fragment developer and since it was not created by composition it is a fragment diagram of level 1. In order to compose network components like an EthernetIPInterface, we have to create a composition program on level 2.

Figure 6.2 depicts the composition program for an EthernetIPInterface. We had to create an empty composition program and add the fragments we want to compose. There is one special fragment, though. The Core is a receiving fragment which only has a receiving port but no configuring ports. It serves as the fragment which all the other fragments are composed into. Hence, it is the only fragment with a receiving port.

Reuseware is designed in a way so that all modeled fragments are stored in a fragment repository which can be browsed in the corresponding view (cf. Figure 6.3(b)). The store folder in the package explorer (cf. Figure 6.3(a)) is marked as Fragment Store which means that the fragment diagrams it contains are added to the fragment repository. Deactivating the fragment store would remove the fragments from the repository. However, what we have to do is select the fragment we want to reuse and press the “+” button to add it to the composition program. This button is highlighted by a red circle in Figure 6.3(b). Since, in our example, there is only one fragment that has a receiving port, contributing and receiving port are linked automatically. These links are indicated by the arrow which goes from the contributing to the receiving port. What we have to deal with are the configuring ports. Unfortunately, there are two dashed circles attached to the builtInEthernetHub but only one to the IP fragment. At this point, the user knows what ports have to be linked and if he does not, Reuseware assists in choosing the right ports by marking invalid compositions with a red x.
Since we have a complete and valid composition program now, we can execute the composition. In Reuseware, this is done by saving the composition diagram. The result can be found by either browsing the package explorer or right-clicking the canvas of the composition program editor and selecting “Open Composed”. In the package explorer, the results are stored in an out folder to make a distinction between the fragment diagrams that were modeled by the fragment developer and the compositions. As stated in Section 2.3.2 only copies of the actual fragments are composed. The out folder, however, is itself a fragment store because in order to reuse recently composed fragments, these have to be made accessible to Reuseware.

Figure 6.3: Package Explorer and Repository View

Figure 6.4 shows the default diagram for the composition program from Figure 6.2. Since no layout adjustment algorithm has been specified yet, there is a big chance that the fragment diagrams overlap even though we placed the icons in a way that they are disjoint. However, one adjustment mechanism has been applied even in this result. The location information provided by the TargetDiagramInformationProvider was
used to move the fragment diagrams to the location specified by us. Without that there would have been no rearrangement at all.

To be specific, in this example we did apply a Layout Composition Strategy. We defined different such strategies in the plugin.xml of the layout language\(^1\) and we can add more by extending this file.

```xml
<composer
  id="none"
  comparator="org.reuseware.coconut.fragment.diagrams.gmf.XYPositionComparator"
  merger="org.reuseware.coconut.fragment.diagrams.gmf.GMFMerger">
</composer>
```

Listing 6.1: Layout Composition Strategy

Listing 6.1 shows the strategy none which we used to create the diagram depicted in Figure 6.4. This strategy is lacking the arranger which is why there is no layout adjustment. By changing the strategy in the Properties view we can add a layout adjustment algorithm to our composition. We just have to type in the ID of the composer we want to apply and save the diagram again to execute the composition. The Figures 6.5(a) and (b) show the composed diagrams for the given composition program with additional layout adjustment. Leaving the strategy field empty or providing a non-existent ID does not mean that there will be no strategy applied. In this case the first one is taken as default.

![Figure 6.5](image)

(a) Horizontal Sorting  
(b) Uniform Scaling

Figure 6.5: Composed Diagram (adjusted layout)

In Figure 6.5(a) Horizontal Sorting is used for layout adjustment and in Figure 6.5(b) Uniform Scaling\(^2\). In both cases the composed diagram does not look exactly like the corresponding composition program. Nevertheless, these results are better than the one depicted in Figure 6.4 since there are no overlapping nodes anymore (Section 3.2, Goal 1), the neighborhood relationship is kept (Section 3.2, Goal 2) and

---

1\footnote{For example, in the package org.reuseware.coconut.fragment.diagrams.gmf}

2\footnote{In this example we work with a scale factor \(s=2\).}
both diagrams are still quite compact (Section 3.2, Goal 3). The user can now decide which one he likes best and change the algorithm accordingly.

Figure 6.6: Level 3 Composition Program

Once this result’s layout is satisfying, we can proceed generating our network infrastructure. Assume we want to build a system and use the ethernetIPInterface. Since the composed diagram for this interface was stored in the fragment store—specifically in the out folder—reusing it is no problem. The steps required for creating a composition program do not change even if we now work on level 3. What we cannot influence on this level is the layout of the diagrams from lower levels. However, there are some things we can influence now. Figure 6.6 shows the composition program which includes the ethernetIPInterface. Navigating through the fragment hierarchy is quite easy once we modeled such a diagram. By double-clicking the ethernetIPInterface icon we can see its diagram and a click with the right mouse key offers the option to “Open Composition Program”. It is useful to have this access to the composition diagram if you want to change the layout composition strategy of low level fragment diagrams. Since this strategy can be set for each composition program individually the composed diagram might look strange. The layout of fragment diagrams which are results of a composition on a lower level will not be adjusted again on the higher level. We decided on treating composed fragments as a whole because we think they must not be rearranged in order to preserve the Mental Map of the user.

Figure 6.7 shows the composed diagram of the given level 3 composition diagram with the Horizontal Sorting adjustment algorithm applied. It is easy to recognize the ethernetIPInterface diagram we just created as well as the adjustment algorithm we applied to it. There is no reason, why we decided on Horizontal Sorting here because there is no big difference to Uniform Scaling in this example.
However, on level 3 there is a difference between the two algorithms. Figure 6.8 depicts the result for Uniform Scaling which is not compact anymore while the diagram adjusted with Horizontal Sorting obeys to that goal. Sorting the diagram horizontally, however, enlarges the diagram in x direction which causes the ADSLStaticIPInterface fragment to be moved the long way instead of being moved in y direction. We already discussed that problem in Section 4.1. Not only is this a waste of space but it also hardly obeys the second goal, preserving the neighborhood relationship. What was meant to be the southern neighbor of the ethernetIPInterface is now moved to the south-east.

In contrast to the layout generated by the Horizontal Sorting algorithm, the example for Uniform Scaling keeps the neighborhood relationship. As depicted in Figure
6.2. **TOPCASED EXAMPLE**

6.8, however, the main problem of Uniform Scaling is the waste of space. Although we used only a small scale factor ($s=2$), the diagram is too large to fit the screen at 100%. Only by zooming out to approximately 40% of its original size, the whole diagram can be seen.

![Diagram](image)

Figure 6.9: Changed Composition Program

Changing the composition program, however, can yield better results. The program depicted in Figure 6.9, for example, creates a better layout for Horizontal Sorting than for Uniform Scaling. Although, again the difference is not too big, Horizontal Sorting best obeys the three goals for layout adjustment while Uniform Scaling yields a diagram that again does not fit the screen at 100%.

### 6.2 TOPCASED example

In Section 6.1 we presented an example for a domain-specific modeling language based on the CIM metamodel and a domain-specific layout technique. Furthermore, we had to adapt the adjustment approach to the GMF layout language. UML class diagrams, activity diagrams, state machines and so on were not mentioned so far. Since these are basic tools for software development to every software engineer we applied the layout adjustment to them as well.

This section features an example taken from the CBSE course of the TU Dresden. In an exercise, the students were to model the software for a pizza shop by using standard UML models and a component editor based on the TOPCASED layout language.
6.2.1 Class Diagrams

Figure 6.10: PizzaShop Composition Program

Figure 6.10 shows a simple composition program for a pizza shop from the CBSE example. On first sight there are at least two differences to be noticed. First of all, there is a fragment icon with gray background color that is not named “Core”. The next thing to notice is that there are no configuring ports to be seen in this composition program. Both these differences are caused by another approach used in this example. The layout of UML diagrams differs and therefore, needs adapted design algorithms for each kind of diagram. In the CIM example we had a blank canvas which all the fragments were merged with. Additionally, the fragments were modeled in a way that they were already distinct of each other regarding the layout and connecting them was realized by drawing new edges between the defined ports. Adding new behavior to an UML Activity Diagram, however, requires insertion of new activities into the diagram while existing transitions need to be kept. Unless a new class is a leaf this has to be done for class diagrams as well. Associations between the new class and the old diagram have to be drawn or rerouted.

In Figure 6.11(a) the class diagram of the PizzaShop is depicted. Highlighting the class Hook_Pizza in that figure emphasizes its special role in this model. This is the location where the class Pizza—depicted in Figure 6.11(b)—will be woven in. Reuseware enables the user to define own composition systems where he can declare how the framework recognizes variation and reference points. “Hook_”, in our case, is the prefix which marks the diagram elements that are the receiving parts (variation points) of the model. Since we already configure PizzaShop.uml by inserting an element via contributing and receiving ports, an actual connection between configuring ports is not required here. Actually, this is the reason for the lack of a Core fragment in the composition program of Figure 6.10. This empty core is not empty anymore but it is one of the fragments involved in the composition process.

Since we first concentrated on the CIM example, we did not think of the option and, hence, when we applied our algorithms to these compositions, unexpected results were generated. Our whole approach is mainly based on taking the layout information from the composition program and use it for the layout of the composed diagram. However, in the composition program we have no option to nest the nodes which is why they will always be regarded as distinct fragments. Without knowing about the structure of the fragment diagram the user would be forced to blindly arrange the icons in the composition program. He cannot create a Mental Map since the standard UML diagrams have a completely different layout than he is able to construct with
6.2. TOPCASED EXAMPLE

Figure 6.11: Class Diagrams of the PizzaShop

the GMF editor provided by Reuseware.

Figure 6.12: Composition Results

Figure 6.12 depicts the results for the composition with our layout adjustment algorithms applied. Although we avoided overlaps in Figure 6.12(a), the layout generated using Horizontal Sorting is not quite what the user would expect. Normally, the composed class diagram should look like PizzaShop.uml just with the hook element replaced by the Pizza class. Furthermore, since the contributing diagrams are checked for overlaps against the bounding box of the receiving diagram, Pizza.uml is not inserted on the western but moved to the eastern side of PizzaShop.uml. However, in contrast to the result from Figure 6.12(b), it does remove overlaps.
Applying Uniform Scaling does not perform any layout adjustment at all. This can be explained by the different implementations of the layout algorithms. In the Listings 5.11 and 5.12 the reason for this is shown. We have to recall that Horizontal Sorting sees only one fragment diagram at a time. This current fragment is checked against the receiving diagram and if there were any overlaps the layout adjustment is triggered. Uniform Scaling, in turn, does not care about the receiving diagram but only about the list of contributing diagrams. The Listings 5.14 and 5.15 provide the source code for that. This means that all contributing fragments are merged with the receiving diagram without checking for overlaps.

To take this example one step further we now assume our pizza shop employed a new cook who is able to cook lasagne, pasta and pizza. In order to visualize this enhancement in our class diagram, we have to add two more variation points. Namely, these are Hook_Lasagne and Hook_Pasta. Additionally, we require the fragments Lasagne.uml and Pasta.uml which will be inserted into the pizza shop diagram via composition. Each of these fragments only consists of one class node like the Pizza.uml.

In Figure 6.13 the characteristics of the TOPCASED layout language become obvious. Figure 6.13(a) shows the extended PizzaShop.uml and Figure 6.13(b) the layout for the composition with Uniform Scaling. As stated, the receiving diagram is not altered during the composition. But, since there are more fragments involved now an adjustment is performed. TOPCASED diagrams have a canvas which does not grow or shrink dynamically which is why the pizza node in the composed diagram is half-hidden.
6.2.2 State Machines

There is one more example we want to present. In a software development process usually different types of UML diagrams are involved since each type is used to model a particular part of the software. While class diagrams are used to give an overview of the used classes and their relations, state machines, for example, are used to model the behavior of the software.

Assume we want our pizza shop to take orders from clients and, subsequently, bake the pizzas of these orders. Figure 6.14 shows how we can model this behavior using UML state machines. More precisely, we use state machine fragments. The PizzaShopStateMachine.uml covers the whole process of ordering and baking pizzas whereof the state for taking orders is not the state itself but the hook where the TakeOrderStateMachine from Figure 6.14(b) has to be woven in. A composition program for this task is shown in Figure 6.15.

Figure 6.14: PizzaShop and TakeOrder State Machines

Figure 6.15: PizzaShopStateMachine Composition Program

Again, we have a diagram for the receiving diagram which is not the EmptyCore. All states of the Figures 6.14(a) and 6.14(b) that do not have the prefix “Hook_” are shown as contributing ports (black circles) in Figure 6.15. The receiving port Hook_PizzaOrdering is the part of the diagram that will be altered by inserting the TakeOrderStateMachine.uml diagram.

As can be seen in Figure 6.16 composing state machines requires a different approach than the one used on the CIM diagrams. Though the layout algorithms remove overlaps when diagrams have to be inserted into other diagrams, the layout of the
composed fragment is not well-arranged. Furthermore, due to the different structure of TOPCASED diagrams, the Merger had to be adapted. This structure—or better, the restricted canvas—causes the disappearance of fragments in case they are moved beyond the bounds.
Chapter 7

Conclusion

To conclude this thesis we want to reflect the issues of diagram layouting in a model-driven composition process that arose in this work. Our goal was to compose models in a user friendly way using given layout information. We wanted to show that, given two layout languages—one for modeling component diagrams and one for the layout of composition programs—, we are able to compose information of both in order to create diagrams that are easily readable. Therefore, these composed diagrams had to obey to some rules like the disjointness of nodes (cf. Section 3.2 Goals 1 to 3). Realizing this required the introduction of an additional concept. The psychological term of the Mental Map had a big influence on this work since it is the basic principle people subconsciously use when confronted with layout composition.

With the Mental Map setting the rules and the Reuseware Composition Framework being the software this concept had to be implemented in, we had to come up with a new concept of layout adjustment. We had to decide on at least one algorithm for this purpose and, more importantly, reconsider Reuseware’s architecture. Changes had to be made to it as well as extensions. In order to provide the opportunity to further develop this approach in the future we decided to make it as layout language independent as possible. However, some parts of the implementation, i.e. Merger and DiagramInformationProvider, had to be language specific.

In the first section of this chapter, i.e. Section 7.1, the dependence of the layout language is issued as well as the general approach; the layout adjustment. Since we presented two layout adjustment algorithms we discuss their fitness for use as well as suggestions to enhance them. Section 7.2 is dedicated to the Horizontal Sorting algorithm and in Section 7.3 we will discuss the results of Uniform Scaling. Finally, in Section 7.4 we elaborate on future work.

7.1 Language-dependence and general approach

As mentioned in Section 2.3.2 the layout information that can be found in a composition program had been discarded by Reuseware until now. Our approach, however, is based on that very information since we believe that the user made up his mind about the layout before he starts arranging the icons in the composition program. In order to protect the user from loosing his orientation in the composed diagram we
place the fragments at the location that was specified by him. After that, the layout required adjustment due to overlapping diagram elements. For our main example, the CIM network modeling, our approach worked fine. Unfortunately, other models such as UML models are built in a completely different way. The icons in composition programs do not provide information about the layout of the corresponding fragment diagram which might lead to unexpected results. In Section 6.2 we presented UML class diagrams and state machines that had special elements within them. These elements were replaced by other class diagrams or state machines. Composition programs in Reuseware do not provide a preview of the composed diagram and hence, users have to be lucky to place the new fragment correct. Although it removes overlapping elements, our approach would not be the best in case of inserting fragments into other fragments. Here it would be better to take the location of the element meant to replace, insert the new element right there and rearrange the surrounding elements in order to make room for the new one. This rearrangement could be performed by algorithms like the Uniform Scaling with the center of the new fragment being the center point.

Gathering the layout information from the fragment diagrams as well as from the composition programs requires our approach to be layout language dependent to a certain degree. Furthermore, the composed diagrams need a layout language as well to be generated. Hence, as stated in Chapter 5, Merger, Source- and TargetDiagramInformationProvider have to be language specific. Sorting the fragments and arranging them, on the other hand, was realized by components that are independent of the layout language. This gives the opportunity to add new Comparators or Arrangers in the future. As soon as another layout language is chosen, its characteristics have to be paid attention to. Not only do different languages have a different syntax but the diagram structure may also change. This means that Mergers cannot only be written again using the new syntax but they may require further adaptation.

GMF and TOPCASED, as it turned out, have some differences, too. GMF, for example, has an infinite canvas that adapts to the size of the diagram while TOPCASED has a restricted canvas and elements that are beyond it disappear. Another difference was the structure of the diagrams. While, in GMF, we had to move the root node first and the following ones accordingly TOPCASED only required to move one node and the others were rearranged automatically.

### 7.2 Horizontal Sorting

One of the two layout adjustment algorithms in the field of interactive graph visualization presented in this thesis is Horizontal Sorting. In Chapter 6 we were able to discover some advantages of this algorithm. However, there are disadvantages as well and, due to that, we want to discuss possible solutions now.

Horizontal Sorting is a good algorithm when it is applied to diagrams where fragment diagrams are distinct already. With this prerequisite fulfilled, it is possible to move overlapping fragments in x direction without destroying the user’s Mental Map. For small examples, Horizontal Sorting meets all the goals set in Section 3.2. However, when the number of diagrams to compose grows the result becomes wider
and wider which violates Goal 3 (compact design). In some cases it would be easy to solve this problem by using a Vertical Sorting which is depicted in Figure 4.2. Implementing this would involve an adaption of the sorting algorithm, though. The XYPositionComparator shown in Listing 5.10 starts with the element that has the lowest x value but this would not generate the best layout either. Even worse, for the example depicted in Figure 4.2 it would violate Goal 2 because the third fragment would be moved beneath the second. Avoiding this problem can be done by splitting the XYPositionComparator into a XComparator and a YComparator. In order to execute the best algorithm a preview of the result should be offered to the user. One could think of even another algorithm which combines the advantages of Horizontal and Vertical Sorting. Depending on the better result a decision for an algorithm has to be made for each step.

The CBSE example discussed in Section 6.2 is not a good one to apply Horizontal Sorting to. Although overlaps are removed, the result is not satisfying. Since UML models—class diagrams as well as activity diagrams or state machines—have a certain layout users might be confused when new elements are aligned horizontally and associations, transitions or other edges cross the whole diagram. When laying out these diagrams users try to create planar graphs rather than having edges crossing each other. When inserting a fragment into an existing diagram, the solution should be regarded as a completely new diagram where it is not that important to recognize the individual fragments. Taking the example of Figure 6.11 one would suggest to insert the pizza class right at the location of the hook in the pizza shop diagram. Although there will not be any overlaps in the composed diagram we could imagine a bigger pizza element with some more properties and operations which would cause an overlap with the associated classes (leaves). Removing these overlaps could then be achieved by applying the Vertical Sorting.

7.3 Uniform Scaling

Similar to Horizontal Sorting, applying Uniform Scaling to the CIM example yielded the better results compared to the layout generated for the CBSE diagrams. Yet again, providing a preview would improve the acceptance.

The implementation of Uniform Scaling required a different approach than the one used for Horizontal Sorting. As mentioned in Section 4.2 a center point has to be computed from which the fragments are moved away by an amount determined by the distance of the fragment to that point and a scale factor \( s \). This scale factor, however, has to be chosen carefully since, if it is too small or too big, the composed diagrams can grow too big as shown in the Figures 4.4 and 4.5. It does not really matter for compositions with few fragments that do not overlap too much but in case we wanted to compose big systems it would be nice to have a preview of the composed diagram and a scrollbar to adjust the scale factor manually.

Applying Uniform Scaling to the CBSE diagrams did not help much. The way we realized it, the algorithm only cares about the contributing fragments and does not check for overlaps with the receiving fragment. This would be important for the kind of composition we face here, where the receiving diagram is not empty. However,
layout adjustment is only triggered if contributing fragments overlap and only these are rearranged by the algorithm. But even if that was not the case and the receiving diagram was included the result would have equal flaws than the result of Horizontal Sorting which were described in the previous section. A solution to that could be to insert new fragments right at the place of the hook in the receiving fragment. The center of this fragment could then be the center point to Uniform Scaling and all the fragments around it would be moved away to make room.

7.4 Future Work

In this minor thesis we were able to implement a way to compose layout information in a model-driven software development process. We stated that, whichever layout languages given, it is possible to adapt our approach to them with little effort. GMF (for CIM diagrams) and TOPCASED (for UML diagrams) were used in our examples. Furthermore, we applied the layout adjustment algorithms Horizontal Sorting and Uniform Scaling to our diagrams.

Future work, however, needs to show that this approach works fine using other languages. Additionally, since this approach was not yet implemented in a tool other than the Reuseware Composition Framework its universal validity requires proof. Horizontal Sorting and Uniform Scaling, as clarified in this thesis, are not always the best choices for layout adjustment algorithms. However, we showed that it is possible to implement algorithms that are based on different approaches and, hence, future work should either enhance the implemented algorithms—we already proposed some refinements in the Sections 7.2 and 7.3—or find algorithms that better fit the purpose.
## Listings

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Diagram Composer</td>
<td>24</td>
</tr>
<tr>
<td>5.2</td>
<td>Bounds</td>
<td>27</td>
</tr>
<tr>
<td>5.3</td>
<td>FragmentDiagram</td>
<td>27</td>
</tr>
<tr>
<td>5.4</td>
<td>Providing of source diagram information</td>
<td>28</td>
</tr>
<tr>
<td>5.5</td>
<td>Comparation of contributing fragments</td>
<td>29</td>
</tr>
<tr>
<td>5.6</td>
<td>Arranging for MSDArranger</td>
<td>29</td>
</tr>
<tr>
<td>5.7</td>
<td>Arranging for SSDArranger</td>
<td>30</td>
</tr>
<tr>
<td>5.8</td>
<td>Computation of the biggest x value</td>
<td>31</td>
</tr>
<tr>
<td>5.9</td>
<td>Check for nodes of type Note</td>
<td>32</td>
</tr>
<tr>
<td>5.10</td>
<td>Comparison of two FragmentDiagrams</td>
<td>34</td>
</tr>
<tr>
<td>5.11</td>
<td>isOverlap(FragmentDiagram, FragmentDiagram) of HorizontalArranger</td>
<td>35</td>
</tr>
<tr>
<td>5.12</td>
<td>The method sortHorizontally(FragmentDiagram, FragmentDiagram)</td>
<td>36</td>
</tr>
<tr>
<td>5.13</td>
<td>findCenterX(List&lt;FragmentDiagram&gt;)</td>
<td>37</td>
</tr>
<tr>
<td>5.14</td>
<td>isOverlap(List&lt;FragmentDiagram&gt;) of UniformScalingArranger</td>
<td>38</td>
</tr>
<tr>
<td>5.15</td>
<td>The method scaleUnified(List&lt;FragmentDiagram&gt;, int, int)</td>
<td>39</td>
</tr>
<tr>
<td>5.16</td>
<td>mergeGMF(EObject, int, int)</td>
<td>40</td>
</tr>
<tr>
<td>6.1</td>
<td>Layout Composition Strategy</td>
<td>46</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Overlapping Fragments</td>
<td>2</td>
</tr>
<tr>
<td>2.1</td>
<td>EMF Model</td>
<td>4</td>
</tr>
<tr>
<td>2.2</td>
<td>Metalevels</td>
<td>4</td>
</tr>
<tr>
<td>2.3</td>
<td>MOF-compliant modelling</td>
<td>5</td>
</tr>
<tr>
<td>2.4</td>
<td>Ecore Hierarchy [14]</td>
<td>6</td>
</tr>
<tr>
<td>2.5</td>
<td>GMF Notation metamodel</td>
<td>7</td>
</tr>
<tr>
<td>2.6</td>
<td>Difference between semantic model and layout</td>
<td>8</td>
</tr>
<tr>
<td>2.7</td>
<td>Reuseware composition</td>
<td>9</td>
</tr>
<tr>
<td>2.8</td>
<td>(a) GMF example, (b) TOPCASED example, (c) composition diagram example</td>
<td>10</td>
</tr>
<tr>
<td>2.9</td>
<td>composition steps</td>
<td>11</td>
</tr>
<tr>
<td>3.1</td>
<td>Fragment Integrity</td>
<td>14</td>
</tr>
<tr>
<td>3.2</td>
<td>Overlaps</td>
<td>15</td>
</tr>
<tr>
<td>3.3</td>
<td>Planarity Algorithm</td>
<td>15</td>
</tr>
<tr>
<td>3.4</td>
<td>Adjustment Step</td>
<td>16</td>
</tr>
<tr>
<td>4.1</td>
<td>Importance of the order of composition steps</td>
<td>18</td>
</tr>
<tr>
<td>4.2</td>
<td>Drawback of Horizontal Sorting</td>
<td>19</td>
</tr>
<tr>
<td>4.3</td>
<td>Uniform Scaling Algorithm</td>
<td>20</td>
</tr>
<tr>
<td>4.4</td>
<td>Unused space due to Uniform Scaling</td>
<td>21</td>
</tr>
<tr>
<td>4.5</td>
<td>Splitting off of fragment diagrams</td>
<td>21</td>
</tr>
<tr>
<td>5.1</td>
<td>Reuseware Extensions</td>
<td>24</td>
</tr>
<tr>
<td>5.2</td>
<td>Nodes containing additional information</td>
<td>32</td>
</tr>
<tr>
<td>6.1</td>
<td>Representations of a fragment</td>
<td>43</td>
</tr>
<tr>
<td>6.2</td>
<td>Composition Program EthernetIPInterface</td>
<td>44</td>
</tr>
<tr>
<td>6.3</td>
<td>Package Explorer and Repository View</td>
<td>45</td>
</tr>
<tr>
<td>6.4</td>
<td>Composed Diagram</td>
<td>45</td>
</tr>
<tr>
<td>6.5</td>
<td>Composed Diagram (adjusted layout)</td>
<td>46</td>
</tr>
<tr>
<td>6.6</td>
<td>Level 3 Composition Program</td>
<td>47</td>
</tr>
<tr>
<td>6.7</td>
<td>Composed Level 3 Diagram (Horizontal Sorting)</td>
<td>48</td>
</tr>
<tr>
<td>6.8</td>
<td>Composed Level 3 Diagram (Uniform Scaling)</td>
<td>48</td>
</tr>
<tr>
<td>6.9</td>
<td>Changed Composition Program</td>
<td>49</td>
</tr>
<tr>
<td>6.10</td>
<td>PizzaShop Composition Program</td>
<td>50</td>
</tr>
<tr>
<td>6.11</td>
<td>Class Diagrams of the PizzaShop</td>
<td>51</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

6.12 Composition Results ............................................. 51
6.13 Extended PizzaShop Example ................................. 52
6.14 PizzaShop and TakeOrder State Machines .................. 53
6.15 PizzaShopStateMachine Composition Program .............. 53
6.16 Composed State Machines ................................. 54
Bibliography


[14] International Business Machines (IBM). Ecore API.


Confirmation

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

Dresden, July 31, 2009