Electronic Communications of the EASST
Volume 7 (2007)

Proceedings of the Workshop on the Layout of (Software) Engineering Diagrams (LED 2007)

Mental Map and Model Driven Development

Jens von Pilgrim

16 pages

Guest Editors: Andrew Fish, Alexander Knapp, Harald Störrle
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Mental Map and Model Driven Development

Jens von Pilgrim*

*Lehrgebiet Software Engineering, FernUniversität in Hagen
Jens.vonPilgrim@FernUni-Hagen.de

Abstract: In the case of a model driven development (MDD) approach, source models are transformed into destination models during the development process. This leads to a transformation chain, that is, a sequence of models connected via transformations. While so far usually only the domain models were transformed, we introduce an algorithm here to also transform the notation models, in order to create a diagram based on the layout of a predecessor diagram. We illustrate the algorithm on several examples which are visualized by a full-featured editor displaying 2D diagrams on layers in a 3D space.

Keywords: MDD, graph drawing, model transformation, mental map

1 Introduction

In the case of a model driven development (MDD) approach, source models are transformed into destination models (and finally code) during the development process. This leads to a transformation chain, that is, a sequence of models connected via transformations. Since these models describe (different) domains of a system they are called domain models.

Most of these domain models are visualized by diagrams [1], i.e. graph drawings with vertices and edges, such as UML or ER diagrams. When a model is displayed by a graphical editor for the first time, some layout algorithm renders the diagram of the model. Known layout algorithms place the nodes of a graph using optimization criteria, e.g. reducing edge crossings. These algorithms can be applied to any model which can be interpreted as a graph, since only the structure of the graph is needed in order to draw the graph.

When setting up a transformation chain, each model is related to a predecessor model, except for the very first models which are usually created (and drawn) manually. This leads to a problem known from dynamic graph drawing: “The user has built up a so-called ’mental map’ that should be preserved when possible” [Bra01]. In dynamic graph drawing only a single diagram is changed. In a transformation chain two diagrams are related to each other. The diagram of a successor model should resemble the previous diagram in order to preserve the mental map. In other words, the “shape” [ESML91] of both diagrams should be preserved as far as possible in order to make the user’s orientation easier.

When visualizing a domain model, additional graphical information is needed for rendering the model, such as the locations of the nodes to be drawn. Usually this graphical information is not part of the domain, i.e. the domain model does not contain this information. This information

[1] I will use the term diagram in the following to address the visualization of a model.

is stored in an extra model, called notation model. [2]

Using model driven terms, the problem of drawing a graph can be described as setting the properties of a notation model. Figure 1 shows the models involved in a transformation chain.

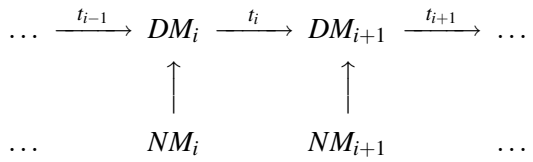

Figure 1: **Transformation chain with domain models (DM) and notation models (NM)**

The first row in figure 1 shows the transformation chain. Some domain models DM_k are related via transformations t_k. Additional notation models NM_k are used in order to display the domain models. To preserve the structure of a diagram, we have to make use of the information stored in the notation model of the predecessor. Therefore, what I suggest here is another transformation t_{NM} mapping the source notation model to a target notation model as shown in figure 2.

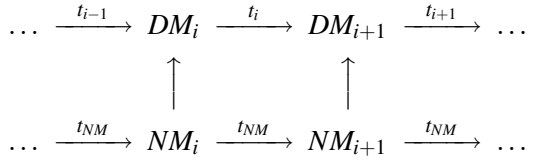

Figure 2: **Transformation of notation models**

While different transformations are needed when mapping the domain models, only one transformation is required for mapping any notation model, since they all use the same notation meta model. As we will see later, the notation model must be initialized by a graphical editor, but the layout algorithm itself is independent from the actual diagrams or models. If the layout algorithm introduced here is implemented by a graphical editor, any transformed model can automatically be layouted based on the source diagram without further effort.

Usually, diagrams in software engineering are drawn as 2D diagrams. One of the most frequently used languages for models is the Unified Model Language (UML), which is also notated using 2D diagrams. Displaying 2D diagrams in different editor windows, it is difficult to compare two larger diagrams, since usually only parts of a diagram can be displayed. Even when we preserve the "mental map", the user cannot really see it on screen. We therefor intend to preserve the characteristic of 2D diagrams, but, at the same time, display multiple (2D) diagrams in a single 3D editor. This enables us to show (and edit) several diagrams simultaneously.

This paper is organized as follows. In the next section, the used models and their meta models are explained. Section 3 introduces an example used later to demonstrate the algorithm. The algorithm itself is explained in section 4. In the following section, a 3D editor is described which was used to implement the demonstration, which is presented in section 6. Section 7 describes related work, followed by a conclusion and directions for future work in the last section.

[2] Sometimes, domain models are also called semantic models, and notation models are called graphical models. In a way notation models are also domain models in the domain of graphical editors.

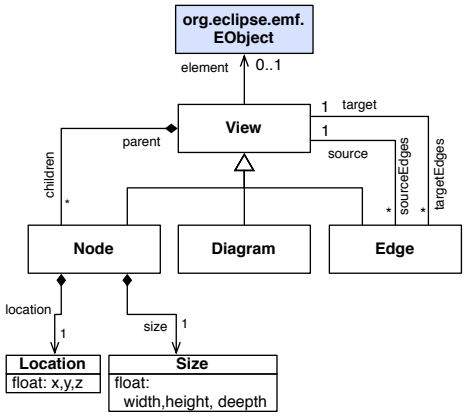

Figure 3: **Simplified GMF notation model**

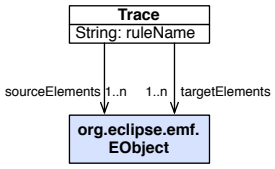

Figure 4: **Simpletrace Model**

2 Models

A model is expressed in terms of a model language. In terms of MDD, a model is an instance of a meta model. UML is a standard meta model for domain models. What we need here is a meta model for notation models. In [OMG06] such a meta model is described (named "Diagram Interchange", short UML-DI). With the wide usage of Eclipse, graphical editors implemented as Eclipse plugins have become very popular. Most of these editors are based on the *Graphical Editing Framework* (GEF) [GEFa], a framework for drawing 2D diagrams, using a Model-View-Control (MVC) (e.g. [BMR+96]) architecture. On top of GEF, the *Graphical Modeling Framework* (GMF) [GMF] is built. GMF consists of two components: a generator framework and a runtime library. While the first consists of model-to-text transformations for generating graphical editors based on some given models, the later also includes a notation model. Since many graphical editors are not only based on GEF but on GMF as well, GMF's notation meta model is becoming more and more popular. E.g. it is used in a number of commercial UML tools such as IBM Rational Sotware Modeler or Gentleware's Apollo, and also in open source projects like the upcoming Eclipse UML2 Tools.

Its notation model is implemented using the Eclipse Modeling Framework (EMF) [EMF]. EMF defines a meta model called "Ecore", a Java based implementation of the Essential Meta Object Facilites (EMOF) [OMG04] [3]. EMF is widely used. We use EMF here for defining all (meta) models. EMF models can be transformed using transformation languages such as the Atlas Transformation Language (ATL) [ATL], which I will use in section 3 when transforming the sample domain models .

The notation model used here is a simplified version of GMF's notation model, its UML class diagram is shown in figure 3. The core of the meta model is a diagram consisting of nodes and edges. It is not a simple graph though, since nodes can contain other nodes, and edges can connect nodes or edges. As in the GMF, domain and notation model are based on EMF. Hence, all classes are derived from EMF's `EObject` class. The link between the notation model and

[3] EMOF is a specification for defining meta models in the context of Model Driven Architecture (MDA) [OMG03], the OMG version of MDD.

the client specific domain model is implemented via the association `element`. This association is directed from notation model to domain model. That is, the notation model is dependent on the domain model and not vice versa. This requires a little bit more effort when transforming the notation, as we have to use a reverse lookup to find a notation element by a given notation element.

When transforming a (source) model into a (target) model, elements of the source model are mapped to elements of the target model. These mappings are called traces. In MDD "everything is a model" [Béz05]. Thus traces are a model, too. In [OMG05], a meta model for traces, called transformation record, is defined. Hence, there may be a standard for traces in the near future using this model. We define a simpler trace meta model here as shown in figure 4. In figure 2 transformations t_k "connected" the domain models. While a transformation is a process, a trace are the result of this process. After the transformation has been executed, a trace connects source model elements with target model elements. A transformation usually consists of rules. Each rule maps a defined set of source elements to target elements. Therefore, we can not only store the model elements but also the name of the rule mapping these elements. Note that we need these traces when transforming the notation, i.e. we need traces of the domain model transformation. Unfortunately, ATL does not create a trace model automatically, but it is easy to implement a transformation which creates traces. Trace generation capabilities can even be added automatically to an ATL transformation as described in [Jou05]. The trace meta model described in [Jou05] is very similar to the model we are using here.

3 Robustness Diagram Example

To give you an impression of the technology on top of which the layout algorithm works, we present here the example which is used in section 6 to demonstrate the algorithm.

We use some kind of robustness analysis diagram [RS99, Jac92] as an example. It is a well known transformation which was used even before MDD has been invented. It maps use cases and actors to classes, which are marked with different stereotypes (Control, Boundary and Entity). It is used to bridge the gap between requirements models such as use case models and design models (usually class diagrams). In order to transform a domain model we need a meta model. For illustration purposes we use a simplified version of UML as shown in figure 5.

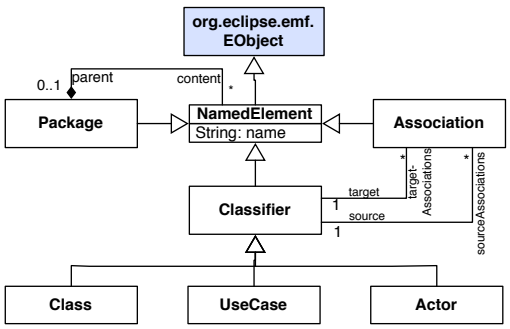

Figure 5: **Simplified UML meta model (Simpleuml)**

```
1  rule UseCase2Class {
2      from
3          usecase: Simpleuml!UseCase
4      to
5          ctrl: Simpleuml!Class (
6              name <- usecase.name + 'C',
7              stereotype <- 'control'
8          ),
9          boundary: Simpleuml!Class(
10             name <- usecase.name + 'B',
11             stereotype <- 'boundary'
12         ),
13         assoc_ctrl_boundary: Simpleuml!Association (
14             source <- boundary,
15             target <- ctrl
16         ),
17         trace: Simpletrace!Trace (
18             ruleName <- 'UseCase2Class',
19             targetElements <- Sequence {ctrl, boundary, assoc_ctrl_boundary}
20         )
21     do {
22         trace.refSetValue('sourceElements', Sequence {usecase});
23
24         -- controller's parent is set in first rule
25         boundary.parent <- thisModule.resolveTemp(usecase.parent,
26                                 'package_classes');
27         assoc_ctrl_boundary.parent <- thisModule.resolveTemp(
28                                 usecase.parent, 'package_classes');
29     }
30 }
```

Listing 1: ATL rule transforming a use case into control and boundary class

In a first test we will use a transformation creating a control and boundary class for each actor and use case. The created classes are connected with associations. Listing 1 shows a rule of the related ATL transformation which transforms a use case to stereotyped classes. The rule, called "UseCase2Class" (line 1), transforms a use case (3) in a control class (5), a boundary class (9), an association between these two classes (13), and a trace (17). The trace is not part of the domain model ("Simpleuml") but of a separate trace model ("Simpletrace") according to the trace meta model described above. ATL is a hybrid language, i.e. declarative parts (lines 4–20) can be combined with operational statements (line 21–28). We need these last statements to solve some special ATL related problems, such as setting a reference to a source model element (without automatically transforming it) (22) and setting references to an element created in another rule (25, 26). It's a 1:3 mapping, since three target elements (Control, Boundary, and Association) are created from one source element; from a layout point of view it is a 1:2 mapping since only one node is mapped to two other nodes.

Figure 6 illustrates the effect of our transformation. The diagrams shown in this figure were actually layouted manually. Figure 7 was added here to give you an impression of the problems mentioned in the introduction. The diagrams in this figure were layouted automatically by a drawing tool (OmniGraffle). Even if better layout algorithms may produce nicer layouts, an algorithm cannot guess the intentions of the developer arranging the nodes. The goal of the algorithm introduced in the next section is to automatically create a target diagram as shown in figure 6.

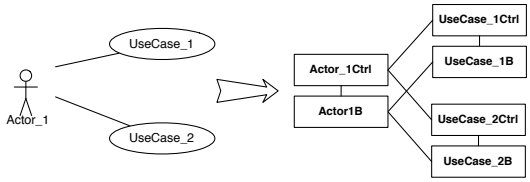

Figure 6: **Use Case Model to Class Model**

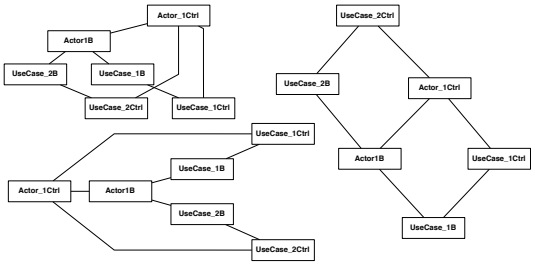

Figure 7: **Three Analysis Diagrams, arranged by OmniGraffle**

4 Layout Transformation

4.1 Assumptions and Constraints

According to figure 2 we have to implement a transformation t_{NM} mapping the source notation model to the target notation model. That is, the layout algorithm presented here is actually a transformation in terms of MDD.

So far, we assume three models to be given: domain model, notation model, and trace model. This trace model is the link between source and target model. But: it is linking the domain models and not the notation models!

As a matter of fact, it is impossible to create a target notation model independently of a graphical editor rendering the model. Even if we have a given notation meta model, it is not possible, since the mapping between domain and notation elements is not defined by the meta model. E.g. imagine two UML class diagram editors (figure 8). One of them shows all classes defined in a model. The other one shows classes of a particular package, classes of other packages referenced by a shown class are listed using text labels inside a package node. If the displayed class model was created by a transformation, we may have a source model and a trace model. But no layout algorithm can create the nodes of the target notation model, because it's the editor which defines what domain elements are displayed as nodes.

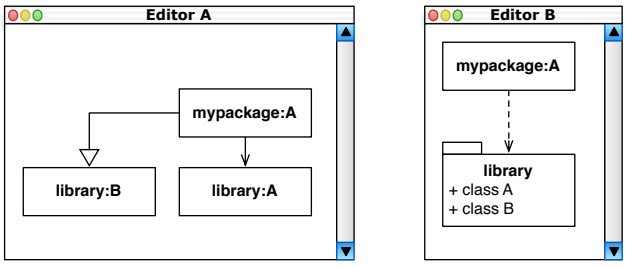

Figure 8: **Two editors displaying the same domain model**

Thus it is the task of the graphical editor to create the target notation model. This is what most editors can do, and some of them use known layout algorithms to set the properties of the notation model. In order to leave our layout algorithm independent from a graphical editor (and

a domain meta model as well), we also assume here that the target notation model is already initialized (but not yet layouted).

The algorithm needs only the two notation models and the trace model. Since it is a layout algorithm, its objective is to render the target elements, that is the elements of the target notation model. The trace contains references to domain elements. The same is true for the notation model (in which the class `View` contains this reference). Essentially, we need to look up the notation elements, i.e. views, by their element. The later is stored in a trace. We therefore do not need to actually access the domain elements, we need only their references as keys for retrieving notation elements.

Before explaining our algorithm, we want to discuss some assumptions and constraints. As a matter of fact, a transformation cannot only map a single source model to a single target model, but m source models to n target models. The sample transformation introduced here maps a single source model to a trace and a domain model. While the domain model transformation can map m to n models, the layout transformation developed here is restricted to a 1:1 mapping, that is, we create one diagram based on one source diagram. Of course we can execute the algorithm m times for m target domain models, but even then we can only use one source notation model. This restriction is valid only if several source models are *not* displayed in a single source diagram. Figure 9 illustrates these cases. This restriction stems from the fact that we cannot merge several notation models yet.

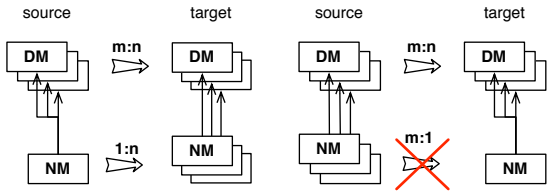

Figure 9: **Algorithm can handle only one source notation model**

The notation meta model as shown in figure 3 allows nesting of nodes. This technique is commonly used in graph editing frameworks. E.g. “attributes” and “operations” of a UML class are often displayed in so called “compartment” nodes, which are children of the “class” node. We therefore have to distinguish between top level nodes, that is nodes which are children of the diagram (or graph, we will use these two terms synonymously) and nested nodes. In the following, we will only refer to top level nodes. That is, our algorithm does not layout nested nodes. Since we only need the location of the nodes of the source notation model, we can also handle nested (source) nodes: We simply use their absolute location, i.e. their location relative to the diagram.

4.2 Layout Algorithm

Figure 10 shows the overall architecture on top of which the algorithm works. In the middle of the diagram you see the `LayoutTransformation` instance. It accesses five models, two of which, the two domain models, are only needed to lookup notation elements via the elements referenced by the traces. Since the trace model references both domain models, the algorithm

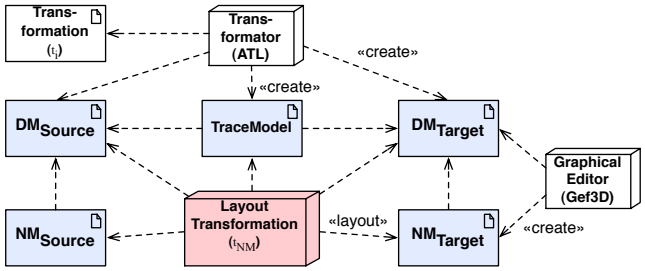

Figure 10: **Deployment Diagram of LayoutTransformation**

only needs the trace and the two notation models as input parameters. A model transformation, e.g. an ATL transformation as used in the example in section 3, is assumed to create both trace and target domain model, while the target notation model was created by the graphical editor.

```
1  public static void layout(Diagram io_diagramTarget,
2      Diagram i_diagramSource, Collection<Trace> i_traces) {
3    Collection<Cluster> clusters = clustering(io_diagramTarget,
4        i_diagramSource, i_traces);
5    arrangeInternalNodes(clusters);
6    adjustClusters(clusters);
7    scale(clusters);
8    declustering(clusters);
9    move(io_diagramTarget);
10   }
```

Listing 2: LayoutTransformation, method layout(...)

Listing 2 shows the overall strategy of the algorithm, each step will be explained more detailed later on. First, *clusters* are created (line 3) [4]. Then, the nodes of each cluster are arranged (5), adjusted (6), and scaled (7). Eventually the nodes are "declustered" (8), i.e. their positions are translated to absolute positions and they are moved to fit into the diagram, that is all locations must contain positive values only (9).

The clusters here are constructed using an *extrinsic* classification: A cluster contains all nodes, the elements of which were created by the same rule with the same source elements. In our implementation, clusters are extended from `Node` (see the notation meta model, fig. 3). Thus, a location can be assigned to the cluster. This location simply is the midpoint of the location of the nodes, which refer to elements in the source domain model. That is, if the transformation simply copies the source model, we have as many clusters as nodes in the source notation model with the same location. We also store the name of the rule, so we can easily retrieve all clusters the node's elements of which were created by the same rule.

Listing 3 shows the classification method. Instead of iterating through the target nodes and assigning each one to a cluster, we iterate through the traces (line 4). This way we do not have to (reverse) lookup the rule (which generated the target nodes) since its name is already stored in the trace. Each domain target element must have been created by exactly one rule, so the union of all traces' target elements equals all elements in the target model. Depending on the graphical

[4] Clusters simply are subsets of nodes of the target diagram (or graph). For definitions and overview of drawing clusters, see e.g. [BC01]. While we use extrinsic clustering, usually the problem of intrinsic clustering is examined.

editor and its presentation of the model, a (top level) node is linked to an element. If at least one node was created in the target notation model (7), we create a cluster (8), store the linked nodes in this cluster (9), and set the location of the cluster to the midpoint of the nodes in the notation source model (11)[5].

```
1  private static Collection<Cluster> clustering(Diagram diagramTarget,
2        Diagram diagramSource, Collection<Trace> traces) {
3      ArrayList<Cluster> clusters = new ArrayList<Cluster>();
4      for (Trace trace : traces) {
5        List<Node> nodes = lookupNodes(diagramTarget, trace
6            .getTargetElements());
7        if (!nodes.isEmpty()) {
8          Cluster cluster = new Cluster();
9          cluster.setInternalNodes(nodes);
10         cluster.setRuleName(trace.getRuleName());
11         cluster.setLocation(lookupLocations(diagramSource, trace
12             .getSourceElements()));
13         clusters.add(cluster);
14       }
15     }
16     return clusters;
17   }
```

Listing 3: Clustering based on source elements and rules

In some cases nodes were created in the target notation model with no corresponding elements. This is the case if the graphical editor displays properties of elements as single nodes. As an example we might think of an icon located close to an element with a special property, e.g. an exclamation mark close to classes with public attributes in an UML diagram. If the node is directly linked to a property of an element, we can resolve its element and process this node as if it is linked to the element directly. This can be achieved in a generic way using EMF's extended reflection mechanism without further knowledge of the domain meta model. In other cases we may add some kind of dynamic graph drawing algorithm (e.g. described in [Bra01] to locate all nodes which were not resolved here. Currently, these nodes are ignored (and located at $(0,0)$).

The nodes are now classified and the created clusters are already located according to the source notation model (fig. 12). In a second step, the internal nodes are arranged "inside" the cluster (Listing 2, line 5). That is, all nodes assigned to a cluster are simply arranged on an ellipse (if more than one node is found). This means that the target nodes are placed on top of each other, as a triangle or as a rectangle and so on (see figure 11).

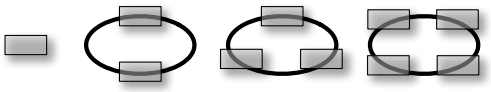

Figure 11: **Arranged internal nodes**

Since we use the ordering of the nodes, which is derived from the ordering of the elements, all elements created by the same rule are arranged in the same order. If, e.g. a model-view-controller triple was created, each role is always located at the same (relative) position, e.g. the controller

[5] lookupLocation automatically calculates the midpoint

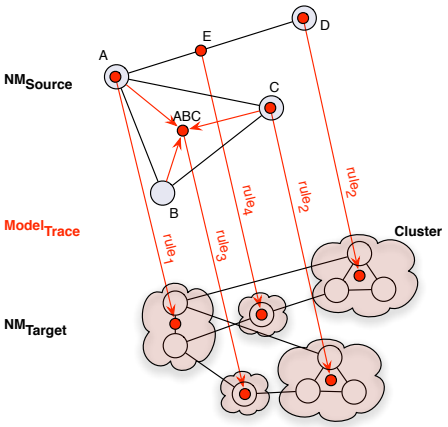

Figure 12: **Clusters**

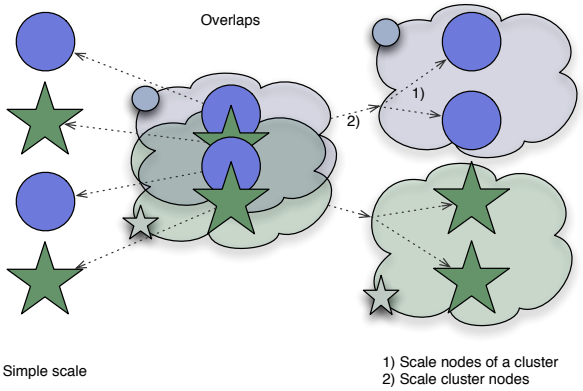

Figure 13: **Scaling all nodes vs. two phase scaling**

on top, the view on the right and the model on the left side. This helps the user to recognize the clusters and the rules afterwards.

To avoid overlaps, we simply scale the diagram. Uniform scaling is a very simple method to eliminate overlaps while preserving the mental map [ESML91]. When using clusters, things get a little bit more complicated. If the diagram is scaled at the end of the layout process, the mental map might be destroyed. This can be caused by nodes of different clusters which overlap in a special way. For example, if a node a of a cluster A overlaps a node b of cluster B, with A is on top of B (i.e. $A.y > B.y$) but a is beneath b. This problem is illustrated in figure 13, where one cluster (shown as cloud) contains circles and the other one stars. To avoid this problem, we scale in two phases: First, the nodes of a cluster are scaled and then, the cluster nodes themselves are scaled. That is, first the nodes of a cluster are scaled in `adjustClusters(...)`, then the cluster nodes are scaled in `scale(...)` (listing 2, lines 6 and 7).

So far we have worked with relative locations, i.e. the locations of the nodes were set relative to the location of their cluster. This means we have to "decluster" the graph, that is the locations of the nodes of the diagram have to be made absolute.

5 Gef3D

Generally, we can use the layout algorithm with any graphical editing framework, because it is only dependent on the notation model. In a model driven approach, models and their relationships are of great importance. That is why we aim at displaying several models at the same time. We achieve this by drawing the 2D diagrams on layers, which are shown in a 3D editor. GEF is a powerful framework for graphical editors. I ported GEF to support real 3D editors and called the result Gef3D [Gefb]. In contrast to many tools for simply visualizing some 3D drawings, Gef3D is an editor with many of features known from GEF.

The main differences between GEF and Gef3D is that a point in GEF is defined as a pair (x,y), where x and y are integer values. In Gef3D a point is defined as a triple (x,y,z) where x, y, and

z are float values. From a users point of view these are the main differences, making it possible to create 3D enabled graphical editors easily if you are familiar with GEF. Things like selecting (called picking in 3D) or moving the camera around (orbiting) are capsulated by Gef3D.

Gef3D internally uses Java3D for rendering the diagrams. Java3D is capsulated and accessed in Gef3D through a Draw2D (GEF's drawing API) like API, now called Draw3D. In fact, Draw3D in Gef3D enables creating 2D or 3D diagrams using a single API. All examples in the following section were all created with Gef3D; all diagrams are completely editable.

6 Demonstration

We implemented a simple Gef3D based editor for drawing use case and class diagrams. We can apply the transformation to a drawn use case model, that is we can generate a class and a trace model automatically. We have four models: source domain model (the use case model), target domain model (the class model), source notation model, and trace model (created by the transformation as well).

We can now use the layout algorithm explained in section 4 to create a notation model for the target domain model automatically. The result is presented in figure 14. This figure shows the source and the target model in a 3D editor, with each model positioned on a (semi-transparent) layer. The source model is placed on top and the target model is visible through the top layer, demonstrating the effect of the transformed layout. The generated diagram equals the manually drawn one of figure 6 – and that is exactly the result that we were aiming at.

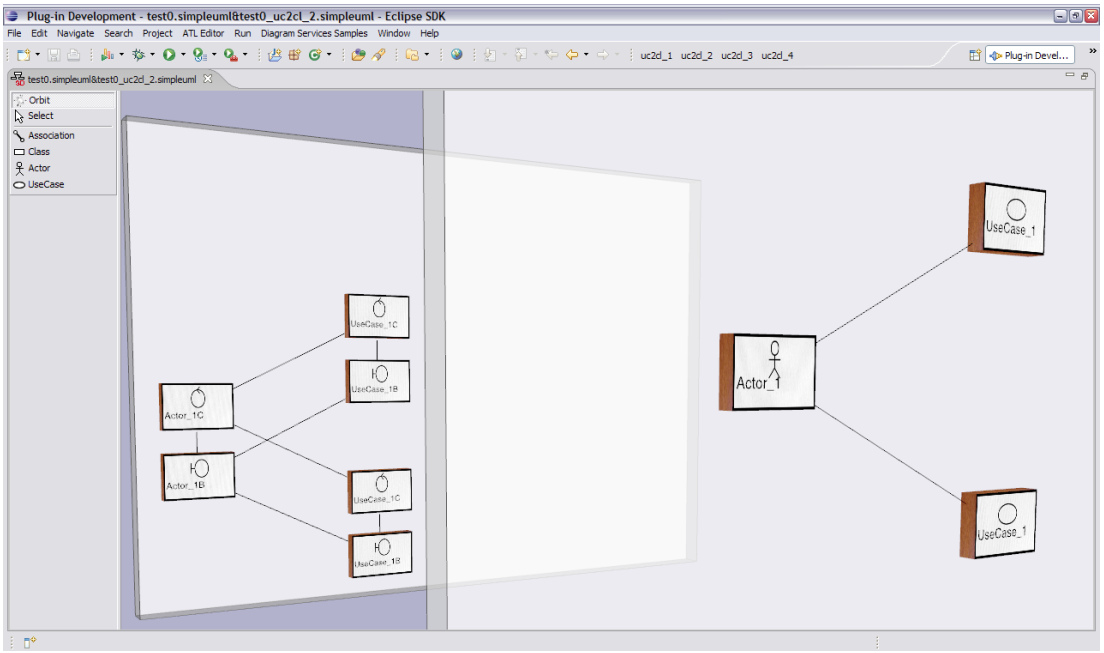

Figure 14: **Target Model (left) and Source Model (right)**

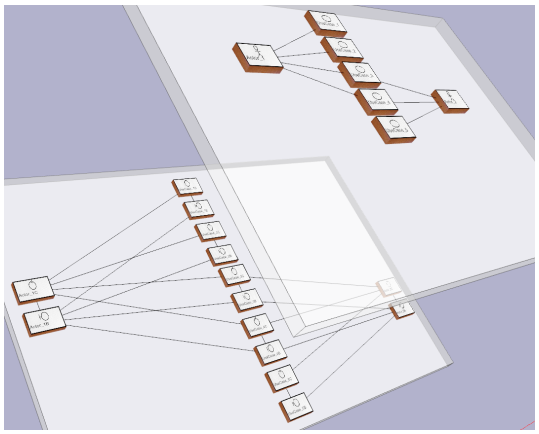

Figure 15: **1:2 mapping, test 1**

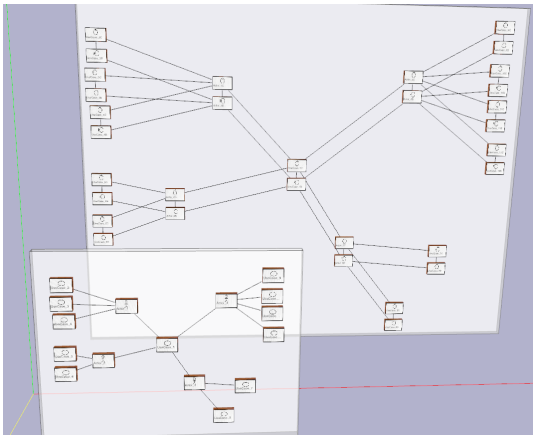

Figure 16: **1:2 mapping, test 2**

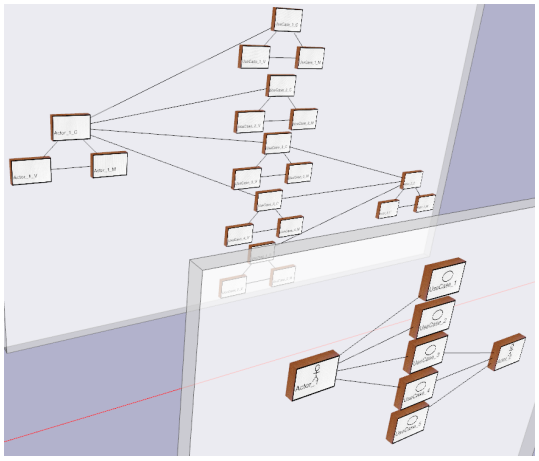

Figure 17: **1:3 mapping (MVC)**

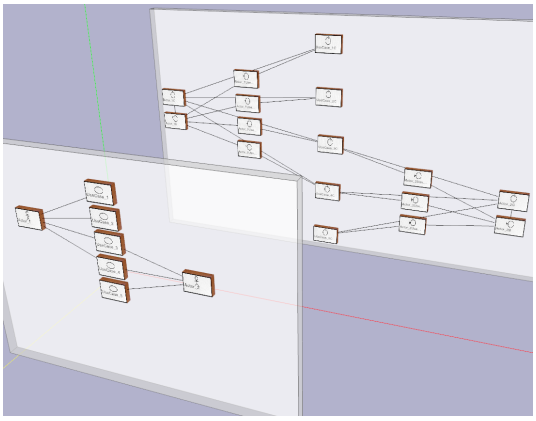

Figure 18: **2:1 (association) mapping**

While this first model is a very simple one, figures 15 and 16 show the same transformation applied on more complicated source models.

It is important to note that the algorithm is independent from any domain model. So, even if we use use cases and classes here, the only thing important for the algorithm is how many source elements are mapped to how many target elements. For testing purposes, we create different types of transformation with different types of mapping (1:1, 1:n and so on). This first transformation implemented a 1:2 mapping. The next example (figure 17) demonstrates a 1:3 mapping with the very same source models. This time, a model-view-control like structure is created.

The transformation of the next example is taken from [SW02], used for teaching software engineering in a student's course. In this case, we create a control class for each use case (1:1), a control-boundary pair for each actor (1:2), and a boundary class for each actor associated with a use case. At first glance, it might seems as if a 2:1 mapping occurs when creating a boundary for an actor-use case pair. But this is not the case, since the association between actor and use case is mapped to the boundary (and two associations). As a matter of fact, mapping the association

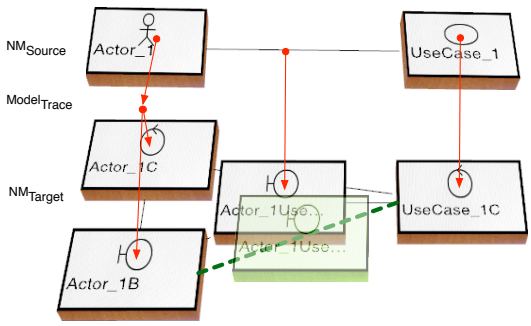

Figure 19: **Problem when positioning nodes created from associations**

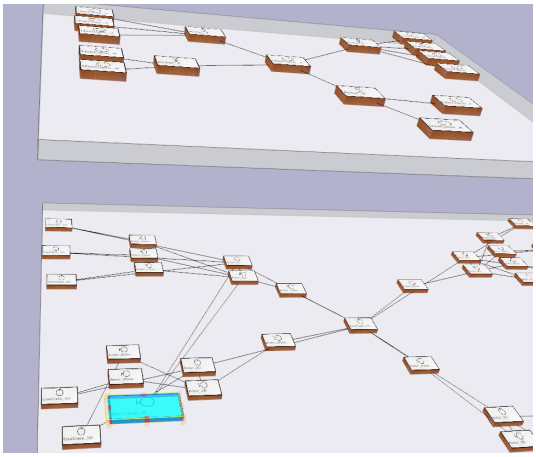

Figure 20: **Finding a bug**

or the pair leads to the same result. Figure 18 shows the result of this transformation applied on the known example.

As you can see from figure 19, a detail of figure 18, this is not an ideal result. Since the boundaries mapped from the association were placed on the midpoints of the associations (in the cluster diagram), but then connected only to one of the two created classes, a lot of overlappings were produced.

By accident we applied this transformation to a buggy source model (i.e. the persistence file of the source model contained a bug due to an error in the editor), so that a wrong element was created. Finding such bugs is usually very difficult. In our case though, the wrong element can be recognized immediately, since the structure of the source and the target diagram differs. The wrong element is selected (and a little bit resized) in figure 20.

7 Related Work

Creating a layout by transforming the layout of a predecessor diagram is, as far as we know, a new approach. But naturally, some of the techniques used in the transformation are well known in graph drawing. Generally, common graph drawing techniques may be applied in order to merge the transformed diagram with manually changed parts. Especially the ideas presented in [ESML91] or techniques used in dynamic graph drawing [Bra01] may be used for this purpose.

Arranging the internal nodes of a cluster, which in our case is achieved by simply positioning the nodes on an ellipse, may be optimized using other graph drawing algorithms as described in e.g. [BETT99] or [KW01]. Maybe a force-directed approach might lead to better results. In any case it is important to arrange all clusters with the same rule in a similar way as described above. This may be added as a constraint to the algorithm.

Visualizing models in a 3D manner certainly has been done before. There are two kinds of 3D visualizations of (graph like) models: Real 3D diagrams and, as in our approach, 2D diagrams on layers in a 3D space. While real 3D diagrams may be useful in certain contexts, we usually have the problem that most standard notations such as UML or ER are defined for 2D diagrams

only. Most approaches are used for program analysis, such as [Vis], [GLW06], or [FBK06]. 2D diagrams on layers were studied e.g. by Gil and Kent [GK98]. They developed different kinds of diagrams for different purposes, such as "contract boxes" for visualizing constraints or 3D sequence diagrams. In contrast to all these approaches I tried not only to implement a visualization tool but to develop a full featured 3D editor instead.

8 Conclusion and Future Work

In this paper I introduced an algorithm for the layout of diagrams based on predecessor diagrams. The algorithm is based on several types of models (domain, notation, and trace model), which are commonly used in the area of graphical editors and transformations. The algorithm was implemented using a 3D editor in order to display source and target model in a single window. The results are quite satisfying, although the algorithm may be improved e.g. when creating nodes based on edges in the source model (fig. 19).

I am currently working on porting Gef3D to directly use OpenGL inside SWT widgets (instead of using Java3D) in order to improve performance and memory usage of the 3D editor. In the context of their final thesis, several students of the FernUniversität Hagen study selected visualization possibilities based on Gef3D, such as the representation of package dependencies, metrics, or traces. We also develop an API for easily implementing layout algorithms with Gef3D without further knowledge of the used notation or domain models. With MDD, models become first class development artifacts and we hope to enable a more intuitive way of working with models as it is possible with 2D editors.

Bibliography

[ATL] Atlas Transformation Language (ATL), Project Website (www.eclipse.org/m2m/atl).
http://www.eclipse.org/m2m/atl

[BC01] R. Brockenauer, S. Cornelsen. Drawing Clusters and Hierarchies. Pp. 193–227 in [KW01].
http://www.springerlink.com/content/hna86wa6ruahmxng

[BETT99] G. D. Battista, P. Eades, R. Tamassia, I. G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, 1999.

[Béz05] J. Bézivin. On the unification power of models. *Software and Systems Modeling* 4(2):171–188, May 2005.
http://springerlink.metapress.com/openurl.asp?genre=article&id=doi:10.1007/s10270-005-0079-0

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. *Pattern-Oriented Software Architecture*. Wiley Software Patterns Series 1: A System of Patterns. Wiley & Sons, 1996.

[Bra01] J. Branke. Dynamic Graph Drawing. Pp. 228–246 in [KW01].
http://www.springerlink.com/content/0ggdwjkyrr5q8u0k

[EMF] Eclipse Modeling Framework (EMF), Project Website (www.eclipse.org/emf).
http://www.eclipse.org/emf/

[ESML91] P. Eades, K. Sugiyama, K. Misue, W. Lai. Preserving the Mental Map of a Diagram. Research report IIAS-RR-91-16E, Fujitsu Laboratories LTD, Aug. 1991.
http://www.iplab.cs.tsukuba.ac.jp/~misue/publications/techreport/iias-rr-91-16e.pdf

[FBK06] A. Fronk, A. Bruckhoff, M. Kern. 3D visualisation of code structures in Java software systems. In *SoftVis '06: Proceedings of the 2006 ACM symposium on Software visualization*. Pp. 145–146. ACM Press, New York, NY, USA, 2006.
http://doi.acm.org/10.1145/1148493.1148515

[GEFa] Graphical Editing Framework (GEF), Project Website (www.eclipse.org/gef).
http://www.eclipse.org/gef

[Gefb] Graphical Editing Framework 3D (Gef3D), Project Website (www.gef3d.org).
http://www.gef3d.org

[GK98] J. Gil, S. Kent. Three Dimensional Software Modelling. In *20th International Conference on Software Engineering (ICSE'98)*. P. 105. IEEE Computer Society, Los Alamitos, CA, USA, 1998.
http://doi.ieeecomputersociety.org/10.1109/ICSE.1998.671107

[GLW06] O. Greevy, M. Lanza, C. Wysseier. Visualizing live software systems in 3D. In *SoftVis '06: Proceedings of the 2006 ACM symposium on Software visualization*. Pp. 47–56. ACM Press, New York, NY, USA, 2006.
http://doi.acm.org/10.1145/1148493.1148501

[GMF] Graphical Modeling Framework (GMF), Project Website (www.eclipse.org/gmf).
http://www.eclipse.org/gmf

[Jac92] I. Jacobson. *Object-Oriented Software Engineering: A Use Case Driven Approach*. acm press. Addison-Wesley Professional, 1992.

[Jou05] F. Jouault. Loosely Coupled Traceability for ATL. In *Proceedings of the European Conference on Model Driven Architecture (ECMDA) workshop on traceability, Nuremberg, Germany*. Nov. 2005.
http://www.sciences.univ-nantes.fr/lina/atl/www/papers/ECMDATraceability05.pdf

[KW01] M. Kaufmann, D. Wagner (eds.). *Drawing Graphs: Methods and Models*. Lecture Notes in Computer Science 2025. Springer-Verlag, 2001.
http://www.springerlink.com/content/xkru1gvnyh5p

[OMG03] OMG. MDA Guide Version 1.0.1. Object Management Group, Needham, MA, omg/2003-06-01 edition, June 2003.
http://www.omg.org/cgi-bin/doc?omg/03-06-01

[OMG04] OMG. MOF 2.0 Core Specification. Object Management Group, Needham, MA, ptc/2004-10-15, convenience document edition, Oct. 2004.
http://www.omg.org/cgi-bin/doc?ptc/2004-10-15

[OMG05] OMG. A Proposal for an MDA Foundation Model. Object Management Group, Needham, MA, ormsc/05-04-01 edition, Apr. 2005.
http://www.omg.org/cgi-bin/doc?ormsc/05-04-01

[OMG06] OMG. Diagram Interchange Specification. Object Management Group, Needham, MA, formal/06-04-04, version 1.0 edition, Apr. 2006.
http://www.omg.org/cgi-bin/doc?formal/06-04-04

[RS99] D. Rosenberg, K. Scott. *Use Case Driven Object Modeling with UML. A Practical Approach.* Object Technology Series. Addison-Wesley, 1999.
http://www.iconixsw.com/UMLBook.html

[SW02] H.-W. Six, M. Winter. *Software-Engineering I. Grundkonzepte der objektorientierten Softwareentwicklung*. Kurs 1793. FernUniversität in Hagen, Hagen, Germany, sommersemester 2005 edition, 2002.

[Vis] ViSE3D (3D Visualization in Software Engineering), Projec Website (http://ls10-www.cs.uni-dortmund.de/vise3d/).
http://ls10-www.cs.uni-dortmund.de/vise3d/