INSTITUTE FOR COMPUTER SCIENCE
LUDWIG–MAXIMILIANS–UNIVERSITY MUNICH

Project Thesis

A MagicUWE extension for semi-automatic layout adjustments of presentation models

Björn Cullmann

Supervisor:	Prof. Dr. Martin Wirsing
Tutor:	Christian Kroiß
Submission Date:	10. August 2010

Contents

CHAPTER 1

Introduction

1. Abstract

UWE is an UML-based approach to model web applications. While different project and diploma theses dealt with migrating this design to the modeling tool MagicDraw by implementing the plugin MagicUWE in Java, this thesis introduces TidyDiagram, a plugin which provides algorithms and methods to implement layout adjusting and grouping actions for UWE's Presentation Diagram.

2. Motivation

The structural part of a web application's frontend can be modeled in a Presentation Diagram (see figure 1). While the layout of normal UML diagrams does not have an influence on the application, the layout of Presentation Diagrams is of utter importance for the development, since the generated web pages will have the same basic structure. Therefore, an element that is modeled to the left of another element, will preserve its position in the application.
Since MagicDraw does not provide the necessary ordering or grouping functionality for this kind of work, this gap had to be closed by TidyDiagram to make modeling a frontend much more comfortable. To give an example, TidyDiagram will provide actions to create groups and to position elements as can bee seen in figure 1 on page 3 below.

3. Goals

Like mentioned above, MagicUWE had to be upgraded with different methods to alter the layout and the structure of Presentation Diagrams. Consequently, TidyDiagram had to be able to:

- order elements by their location in the diagram in a row layout, with an editable number of elements in one row. Optionally, all elements can be enlarged to have the same size.
- Create a new PresentationGroup around existing elements in the diagram
- delete an existing PresentationGroup without losing any of its children by moving them to the parent of the deleted group
- validate the structure in the diagram against the structure in the model. Optionally, the model should be aligned to fit the diagram's structure.

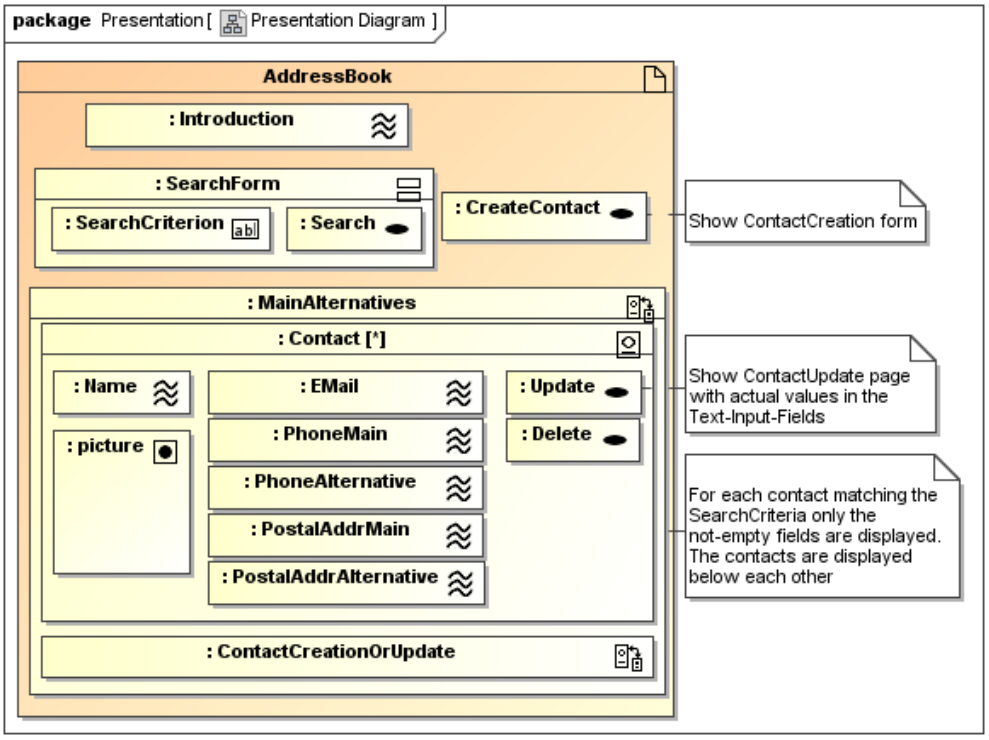

FIGURE 1. A typical Presentation Diagram, representing the structural layout of a web page. The groups MainAlternatives and SearchForm have well positioned children.

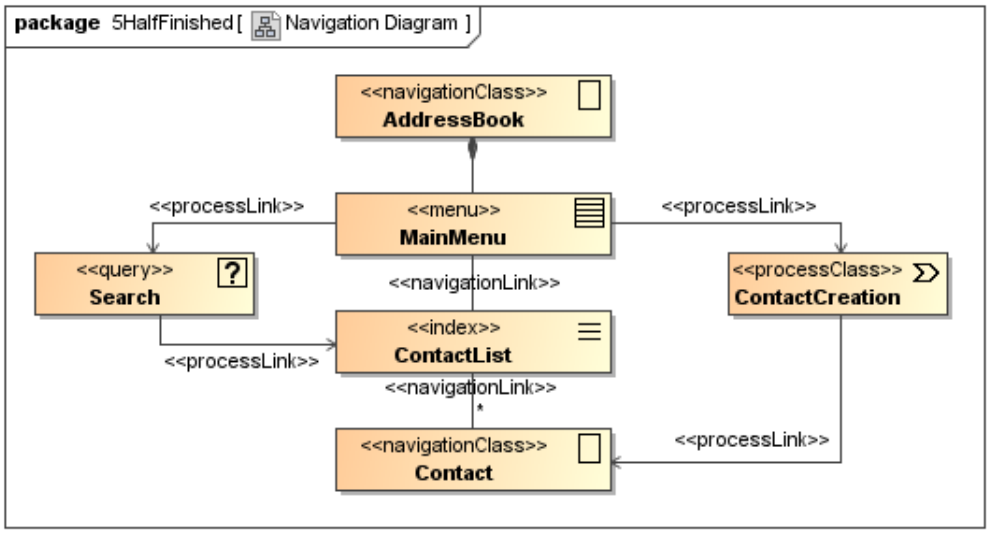

FIGURE 2. A typical Navigation Diagram, representing the connections between the elements.

CHAPTER 2

Background

1. What is UWE?

UWE (UML-based Web Engineering) is an engineering design to provide a domain specific notation, a model-driven development process and a tool support for the web domain. One of the main reasons why this approach is UML-based, is because of UML's acceptance in industry and science as well as its flexibility, making it possible to define so called UML profiles. Profiles are extensions for customizing UML models to adapt to the special domain they are used in. The UWE profile introduces five new views on the project in order to help the developer design a web application. These views cover different aspects of a web project: First, there is the content model which lists all used classes of our project. Then, the navigation model represents the relationships (hyperlinks) between our different nodes (parts of the website). The presentation model defines the structural layout and special HTML elements like forms for our website. The last two models deal with the different processes of our web application: the process structure model describes the relation between different process classes and the process flow model specifies the activities connected with each process class.

The view that will be important for TidyDiagram is the presentation model which is represented in MagicDraw as a composite structure diagram. The top level elements in this diagram are classes, while the nested elements are so called attributes. Therefore, in figure 3, AddressBook is the top class and all nested elements are instances of their according classes. Since instances can have attributes on their own, SearchForm can hold the attributes SeachCriterion and Search. The diagram below shows also that every element can be annotated with a stereotype, giving it an extra meaning. These stereotypes do not only improve the understanding of each element, but also are the key parts for code generation.
Another UWE tool called UWE4JSF[1] implements this generation for Java Server Faces.

[1]UWE4JSF is an eclipse plugin which allows the automatic generation of web applications, http://uwe.pst.ifi.lmu.de/toolUWE4JSF.html

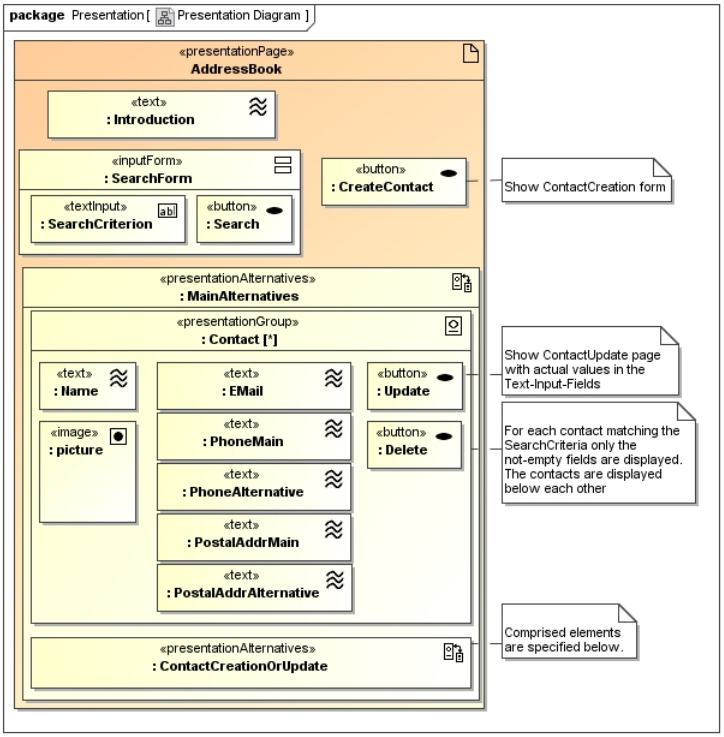

FIGURE 3. Stereotypes can be shown as an icon in the top right corner of each element, or explicitly as a text.

2. MagicDraw

MagicDraw[2] UML is a commercial CASE tool developed by No Magic Inc. The tool supports UML 2.3 standard, code engineering for multiple programming languages (i.e. Java, C++) as well as for data modeling.

2.1. MagicUWE. MagicUWE is a plugin for MagicDraw, providing an implementation of the UWE approach. Due to the plugin, the developer is able to use the UWE profile, which is granting access to its specific stereotypes and diagram transformations, in a comfortable way. Furthermore, MagicUWE performs simple transformations from one view to another. For example, a basic navigation can be derived from the content model.

2.2. MagicDraw's OpenAPI. MagicDraw provides an OpenAPI for plugin development in Java. This API provides a rich library that gives you access to the GUI as well as the UML model. To support developers, an user guide, as well as the javadoc of the libraries are available in MagicDraw's installation folder. While the documentation provides a basic understaning of the whole subject, the developers can give more specific information in the website's forum.

[2] http://www.magicdraw.com

2.3. MagicDraw's Internal Data Structure. Since our plugin will provide actions for aligning and rearranging the diagram elements, it is crucial to have a good understanding of the relationships between the `PresentationElements`. Every UML model element can be represented in a diagram by an instance of `PresentationElement`. `PresentationElement` itself is only used as a parent class for the class `ShapeElement`, which again is parent of different View classes, as cann be seen in the diagram below. MagicDraw uses different Views to represent the model elements. For example, if the model element is a class, it is represented by a `ClassView` object. This object again has different kinds of Views as children, which represent different parts of the visible element.

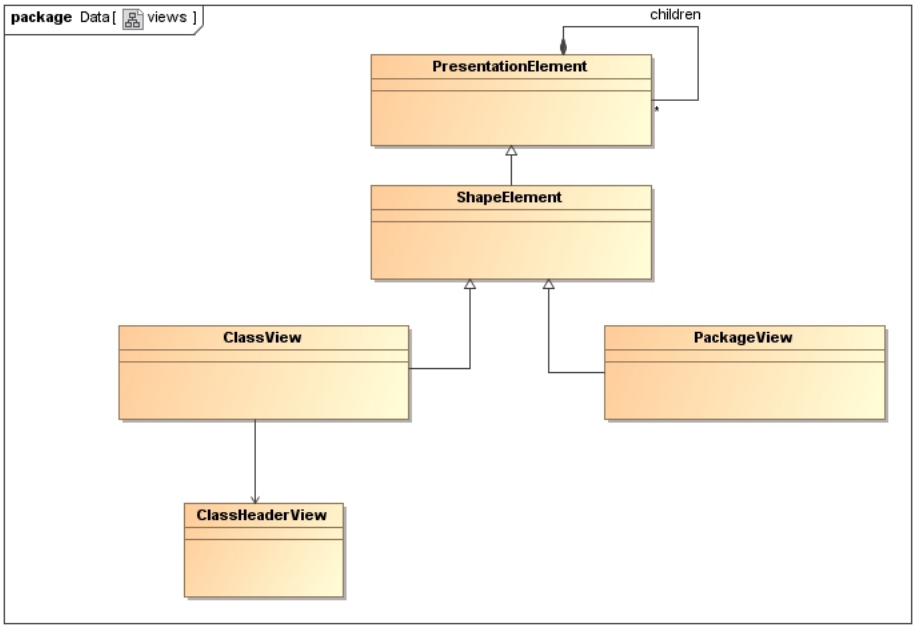

FIGURE 4. A `ClassView` is a `ShapeElement` that holds three different Views on its own.

The most important child of `ClassView` is the `ClassHeaderView`. It has twelve children, six of those are so called `CompartmentViews` which can have children of their own. The most interesting CompartmentView is the `StructureCompartmentView`, because it contains all children of our class.
Every model element has a slightly different structure. A package or a property has its own Views (`PackageView`/`PartView`) and also some special CompartmentViews, but most of the structure, especially the `StructureCompartmentView`, is the same. While the user guide gives a short overview of the class hierarchy down to the `ShapeElement`, it lacks any mention of the class View. The JavaDoc doesn't list

any of the Views mentioned above. The usage of undocumented classes and methods is not recommended by the developers[3], but without these in-depth alterations the given goals could not be achieved.

This knowledge was gathered by the `AuxiliaryFunctions` class (in chapter 3 section 6), which was created for this purpose only.

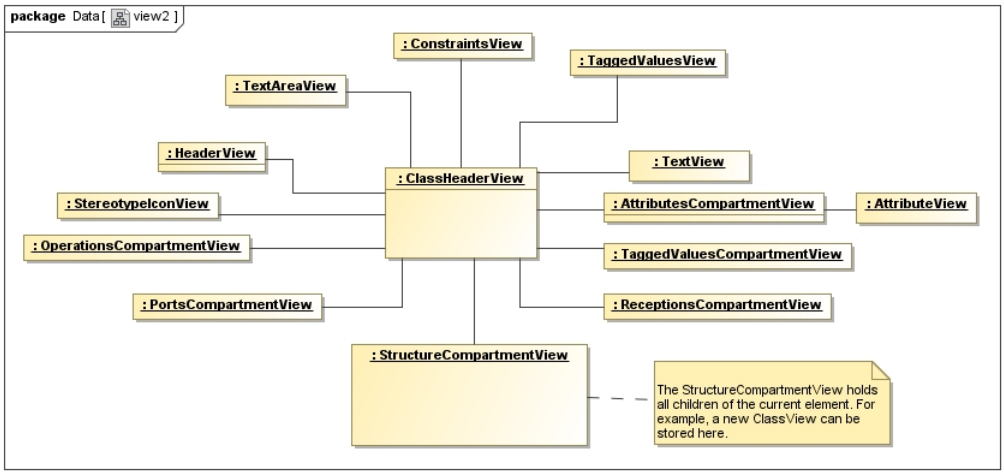

FIGURE 5. A typical configuration of the `ClassHeaderView`, which acts as a container for other Views.

[3]NoMagic Community Forum, https://community.nomagic.com/viewtopic.php?f=26&t=674&p=1828&hilit=only%20documented#p1828

CHAPTER 3

Implementation

1. Architecture

This section gives a short overview of the TidyDiagram plugin. As illustrated in the diagram below, there are three packages: While the main package provides all necessities to integrate this plugin in MagicDraw and additionally a library for often used methods, the actions package contains all actions that implement the real functionality, as well as two supporting classes. At last, the treeStructure package realizes a tree-based data structure for the grouping actions (in section 4.3).

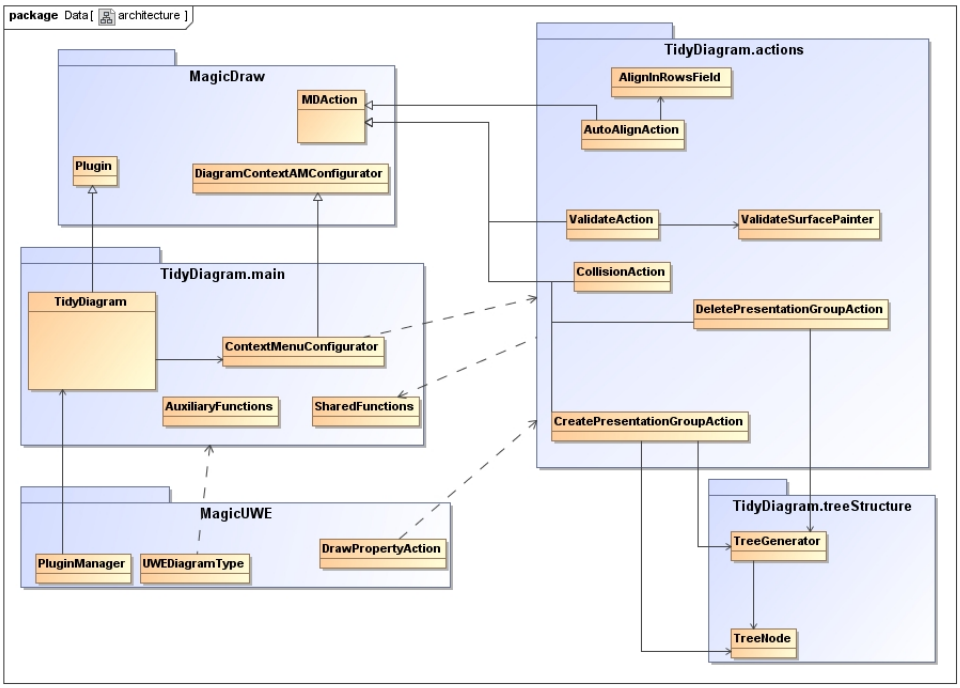

FIGURE 6. Class diagram of all TidyDiagram classes.

Furthermore, this short description will provide a better understanding of the mentioned classes and packages:

Package main

- **TidyDiagram**
  Main Class for the Plugin, it initializes the Configurator
- **ContextMenuConfigurator**
  Initializes all Actions
- **SharedFunctions**
  Library class
- **AuxiliaryFunctions**
  Auxiliary Class for understanding MD's data structure

Package actions

- **Actions**
  Different actions implementing the functionality stated in the requirements
- **AlignInRowsField**
  Realizes a popup, asking for the amount of elements to be sorted in a row
- **ValidateSurfacePainter**
  Paints the order given by the ValidateAction in the active diagram

Package treeStructure

- **TreeGenerator**
  Class to manage the tree structure
- **TreeNode**
  Realizes a single node in the tree

2. Integration in MagicUWE

Let's take a closer look at the relation between MagicUWE and TidyDiagram: In theory, TidyDiagram is an independent plugin that could be run without MagicUWE. Since half of its actions are for UWE's PresentationDiagrams only, the features would be strongly limited, though.

Due to the high encapsulation, the plugins aren't linked tightly. This separation allows both projects to alter and grow whithout having an impact on the other, simplifying future development. In view of MagicUWE, TidyDiagram is just another variable in its `init()` method. This method is called as soon as MagicDraw integrates MagicUWE during start-up. After this initialization, there are no further calls.

On the other side, TidyDiagram is in need of two different classes of MagicUWE. The first and simpler class is the enumeration UweDiagramType that provides the list of possible UWE diagram types, like the PresentationDiagram. This class is used to define actions for the PresentationDiagram only. The second and last class being used is the DrawPropertyAction. In order to create a new PresentationGroup, this class was extended by a method called `createNewPGAndReturnItWithProperty()`. This is very similar to the existing `createClassAndGetProperty()` method but was adapted to explicitly create a PresentationGroup and returning its class as well as its Property.
Beyond these three interconnections, there are none.

3. How To Use TidyDiagram

The TidyDiagram actions can be accessed via the context menu in any diagram. There are two different kind of actions:
The first group will perform the action directly on the selected element(s). This behavior is implemented for the "Handle collisions", "create"- and "delete PresentationGroup" action.
The second group will perform the action on the children of the selected element. Therefore, to auto align all children in the PresentationGroup PG, PG itself has to be selected.
All actions, except for the "create new PresentationGroup" action, insist on selecting only one element. Furthermore, only the first three actions shown in the picture below are available for all diagrams – the other three are for PresentationDiagrams only.

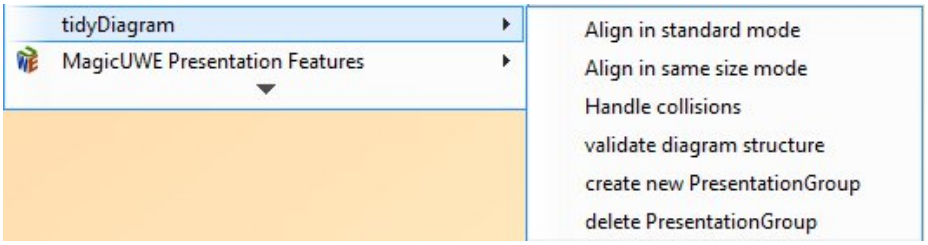

FIGURE 7. Class diagram of all TidyDiagram classes.

4. Implementing the Functionality

4.1. Auto Align. Like mentioned before, a basic requirement for the plugin was to implement a way to order specific elements in a row layout. While this task seems to be quite trivial, some of MagicDraw's customs made it difficult to implement.
First of all, MagicDraw only holds an unordered collection of PresentationElements, therefore it was necessary to implement an algorithm to define the order of elements in the actual diagram. Second, these elements aren't just PresentationElements like the user guide promotes, but a different set of nested and undocumented Views, like mentioned in section 2.3. It is self-explanatory that only the Views that represent a whole model element and not only one part of it have to been found and altered. Given the number of different Views, a filter operation (`recursiveAddChildrenToList()` in section 5) had to be implemented to perform this task.

Still, the main work was to get a better understanding of the structure and functionality of MagicDraw, therefore it was necessary to create a class solely for this purpose. This `AuxialiaryFunctions` class will be described in detail later in section 6.

After understanding the underlying structure, it was necessary to define an algorithm that orders the given elements due to their position in the diagram. The first part was to define the element, that was closest to the relative origin of its parent. Then, the algorithm should search for the nearest neighbor until all elements are processed.

To find the first element, the `getShortestWayToOrigin()` method in the `SharedFuctions` class was created. This method calculates the distance between the relative origin (which is the top left corner of the parent element) and the top left corner of each element that should be ordered. These two points define a rectangular and the Pythagorean Theorem returns the diagonal aka the distance between these points (see figure 8 and Listing 1 below). The PresentationElement with the shortest distance will be returned.

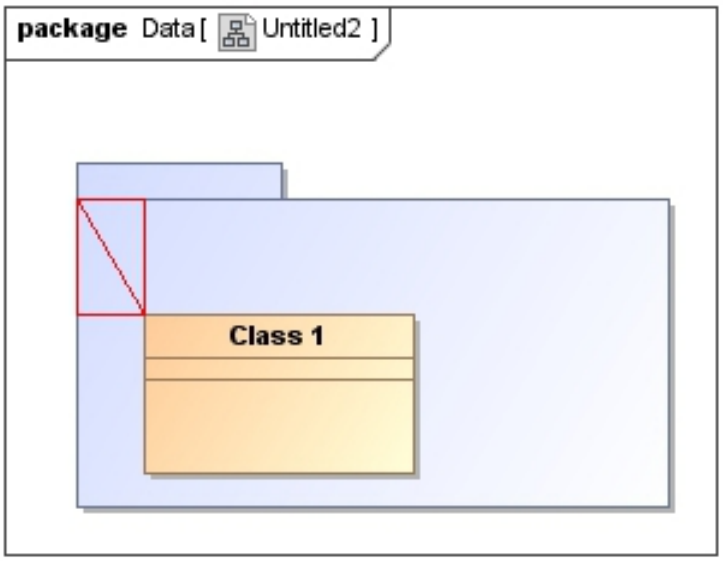

FIGURE 8. `getShortestWayToOrigin()` calculates the distance between each element and the relative origin. The element with the shortest distance will be returned.

```
1  METHOD getShortestWayToOrigin(Collection remainingPresentationElements)
2  {
3          shortestWayToZero: Integer
4          elementWithShortestDistance: PresentationElement
5
6          FOR EACH(currentElement in remainingPresentationElements)
7          {
8                  INITIALIZE fatherElement WITH parent element of currentElement
9                  CALCULATE distanceOfCurrentElement:
10                 {
11                 getRootOf{((currentElement.TopLeftCorner.PositionX - fatherElement.
                         TopLeftCorner.PositionX) * (currentElement.TopLeftCorner.PositionY -
                         fatherElement.TopLeftCorner.PositionY))}
12                 }
13
14                 IF (shortestWayToZero > distanceOfCurrentElement) THEN
15                 {
16                         shortestWayToZero = distanceOfCurrentElement
17                         elementWithShortestDistance = currentElement
18                 }
19         }
20         return elementWithShortestDistance;
21 }
```

LISTING 1. getShortestWayToOrigin() as pseudo code

The next step is to find the element that is closest to our current element. The `getNextNeighbor()` method in the `SharedFunctions` class searches for this element by taking all elements under consideration, which upper left corner is to the top or to the right of our current element. Now the algorithm searches for the shortest distance between our current element's top right corner and the top left corner of every element that satisfies the mentioned condition (see the rectangle in figure 9).

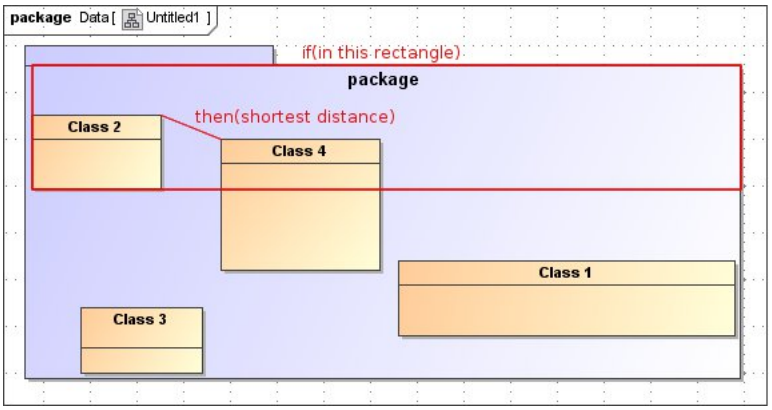

FIGURE 9. `getNextNeighbor()` searches for all elements that are to the top or to the right of the current element. After this, the element with the shortest distance will be returned.

```
1  METHOD getNextNeighbor(Collection remainingPresentationElements, PresentationElement
       lastElement)
2  {
3   nextElement: PresentationElement
4   smallestSpace: Integer
5
6   // check, whether there are other elements in the same "row"
7   FOR EACH (currentElement in remainingPresentationElements)
8   {
9    IF (currentElement's top left corner is to the top or to the right of the lastElement's
         bottom right corner)  THEN
10   {
11    calculate currentDistance:
12    {
13     getRootOf{((currentElement.TopLeftCorner.PositionX - lastElement.TopRightCorner.PositionX)
           * (currentElement.TopLeftCorner.PositionY - lastElement.TopRightCorner.PositionY))}
14    }
15
16    IF(smallestSpace > currentDistance) THEN
17    {
18     smallestSpace = currentDistance
19     nextElement = currentElement
20    }
21   }
22  }
23
24  // if there are no elements left in the same row, the algorithm will search for the first
       element in the next row
25  if (there are no other elements in this row) THEN
26  {
27   nextElement = getShortestWayToOrigin(remainingPresentationElements)
28  }
29  return nextElement;
30 }
```

LISTING 2. getNextNeighbor() as pseudo code

This operation will be performed recursively, always on the last found element, until no element can be found in this row. As soon as this happens, the whole algorithm starts over again, searching for the element with the shortest distance to the origin and fetching the nearest neighbors until all elements have been processed.

```
1  private Vector<PresentationElement> sortElementsByGUILocation(Collection presentationElements)
2  {
3          Vector<PresentationElement> sortedElements = new Vector<PresentationElement>();
4          Vector<PresentationElement> presentationElementsCopy = (Vector<PresentationElement>)
               presentationElements.clone();
5          sortedElements.add(getShortestWayToOrigin(presentationElementsCopy));
6          presentationElementsCopy.remove(sortedElements.get(0));
7
8          while (presentationElementsCopy.size() > 0)
9          {
10                 PresentationElement nextElementToBeAdded = getNextNeighbor(
                       presentationElementsCopy, sortedElements.lastElement());
11                 sortedElements.add(nextElementToBeAdded);
12                 presentationElementsCopy.remove(nextElementToBeAdded);
13         }
14         return sortedElements;
15 }
```

LISTING 3. sortElementsByGUILocation()

To clarify this algorithm, 5 elements will be processed exemplarily:

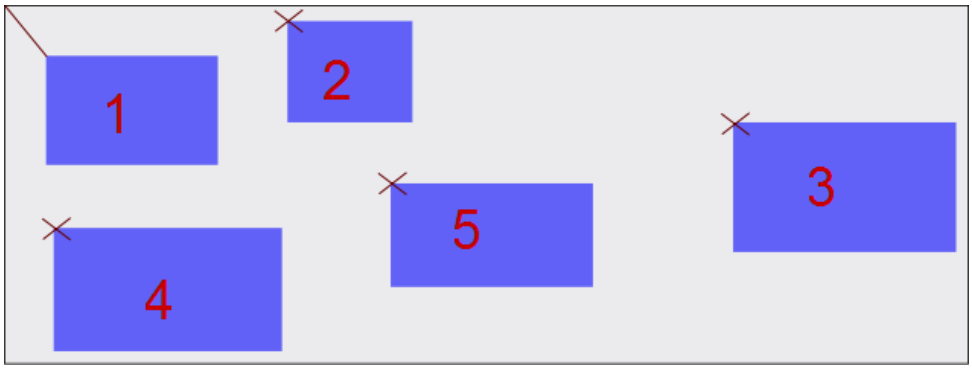

FIGURE 10. The element with the shortest distance to the origin will be picked.

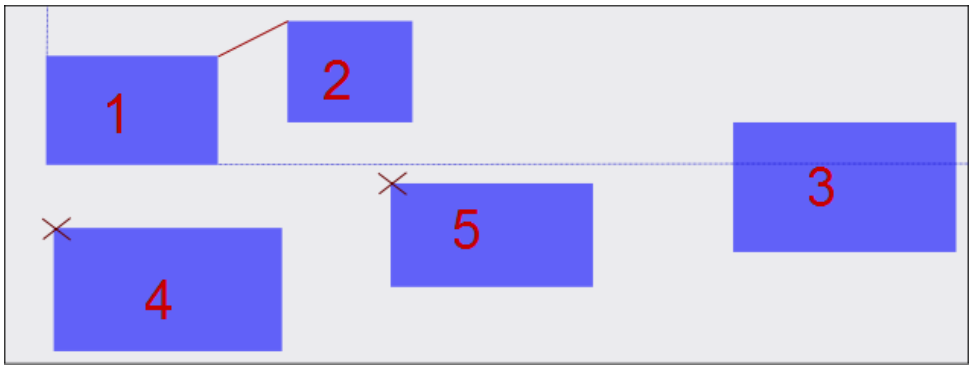

FIGURE 11. The first element opens up a new row, elements 4 & 5 aren't part of it. Therefore the algorithm will check whether element 2 or element 3 is closer to the first element. In this case, it is 2.

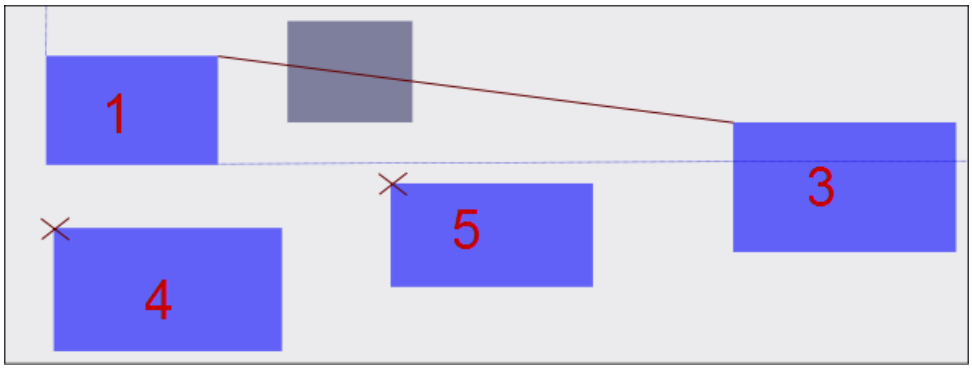

FIGURE 12. Now only element 3 remains in the first row.

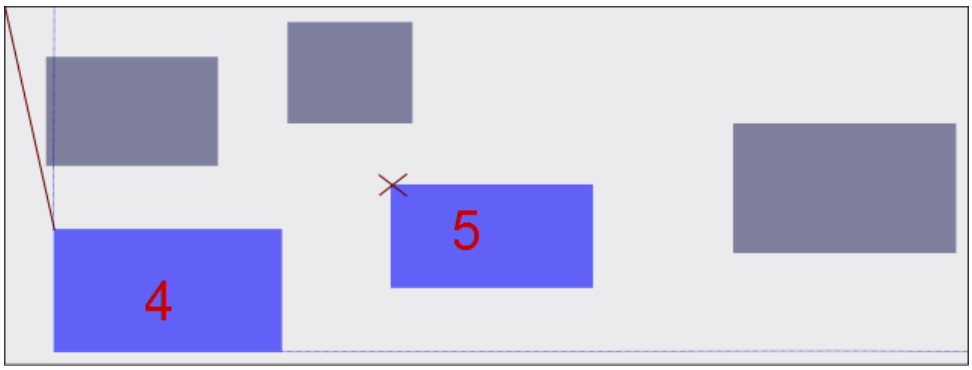

FIGURE 13. After the last row is processed, a new row has to be created. The algorithm starts over again and selects the element closest to the origin.

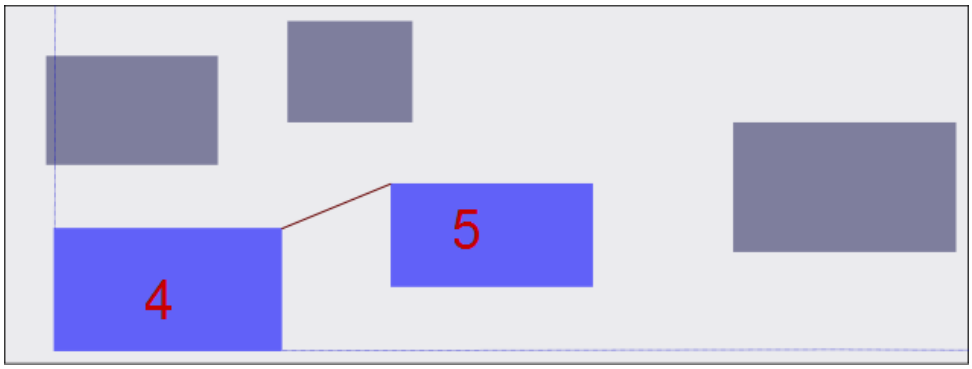

FIGURE 14. Since only one element is left, it will be picked last. The algorithm terminates.

AutoAlign in MagicUWE

After selecting the element whose children should be aligned, the action asks for the amount of elements that should be ordered in one row. This is implemented by the class `AlignInRowsField`, With this information and after filtering the correct PresentationElements, either the standard or the same size mode of this action is called.

4.1.1. *Standard mode.* The standard mode aligns all children elements of the selected element in rows, without touching their bounds. After the given number of elements, the method will order the next set of elements to the bottom of the last row. The height of a row is specified by the element with the largest height.

4.1.2. *Same Size mode.* The same size mode is an extension of the standard mode, extending the size of all elements to the largest height and width. Hence all rows have the exact same height and width, too.

4.2. Collision Handling. After every alteration, it is possible for elements to overlap other elements. To prevent this unwanted behaviour, every action has to check after its completion if there are collisions. To cope with these, an algorithm that solves any collisions intuitively, had to be defined. The algorithm had to rearrange the elements that collided with the altered element.

The main point of interest in this was finding a solution that rearranged the elements in a comprehensible way, so that the elements can be found at a location the user would expect. The second requirement was to rearrange elements in a way that most of the other elements weren't tampered with.

After considering different approaches, a mix of two different styles where chosen.

At first, a check is made whether the altered element covers the other element completely. If this is true, a rather simple test will determine whether the top left corner of the collided element is to the top or to the bottom of the altered element's half height (red line in figure 15). In case one, the collided element will be pushed to the right, in case two to the bottom of the altered element.

To fullfill the second requirement of changing as few elements as possible, only these two directions are implemented, since MagicDraw extends its whiteboard to the right and to the bottom by nature.

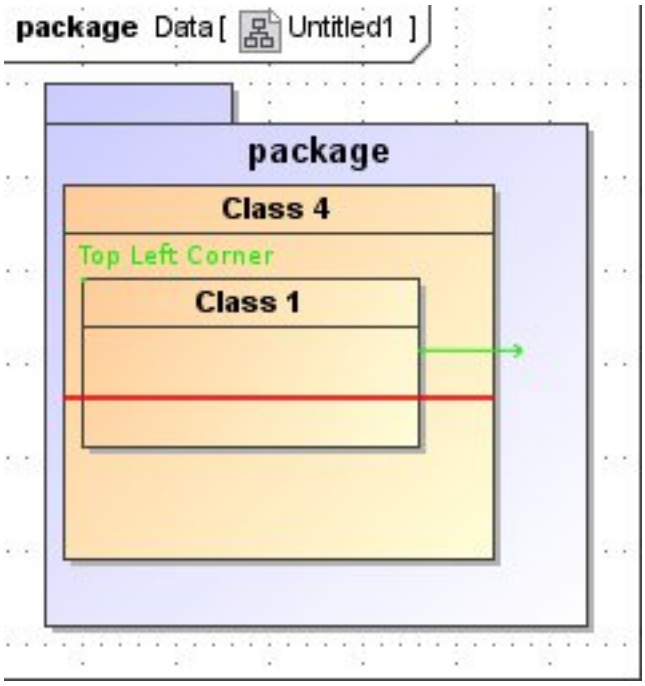

FIGURE 15. The position of the top left corner decides whether the element will be pushed to the bottom or to the right.

If the collided element is only partly covered by the altered element, the algorithm determines whether the rectangle to the right or the rectangle to the bottom is larger. The element will be pushed to the side with the larger rectangle (the green

one in figure 16), leaving the opposite axis untouched. Therefore, if the collided element's x-position is changed because it its pushed to the right, the y-position will be the same.

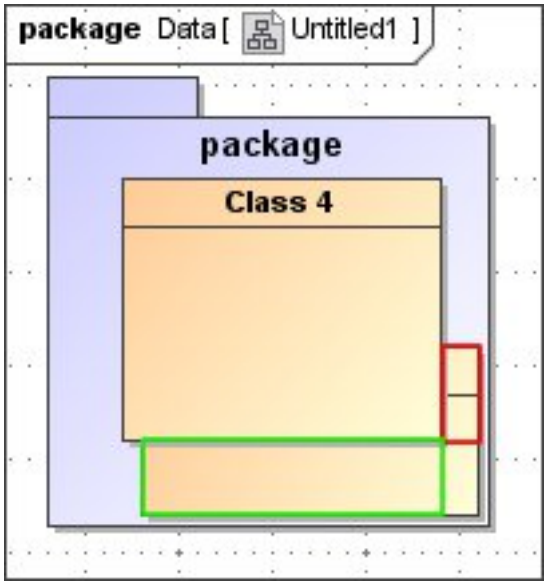

FIGURE 16. The rectangle to the bottom right will not be calculated, since it is a part of the rectangle to the right and to the bottom.

Since every element that has been repositioned can collide again with another element, all realigned elements will be checked for collisions recursively. To avoid unwanted sideeffects, only sibling elements of the same class will be taken under consideration.

4.3. Grouping Actions. To ease working with presentation diagrams, an action to group PresentationElements was necessary.
The first approach for regrouping was to shift the old PresentationElements to their new parent element. This attempt had the result of adding a new attribute to the parent, without having a visible PresentationElement in its compartment. To fix this problem, every PresentationElement had to be recreated in the parent's compartment, ideally with its old bounds. Since the old bounds were lost after any rearranging attempt, they had to be saved before any alteration was performed.
In order to realize this in MagicDraw, a new data structure had to be set up. This data structure had to save all PresentationElements that could be affected by the regrouping as well as their old bounds.

4.3.1. *Tree Structure.* This new data structure was implemented as a inheritance tree, with each node representing a Property's PresentationElement.
A TreeNode realizes this representation, giving access to its PresentationElement and its bounds, both absolute and relative to the parent element. This TreeNode also knows all direct children of its PresentationElement that are Properties.
For initializing TreeNodes, the class TreeGenerator was created. Its purpose was to get an PresentationElement and to create TreeNodes for itself as well as for all its children that are Properties. To be able to navigate through the tree, it also gives access to the root of the newly created tree.

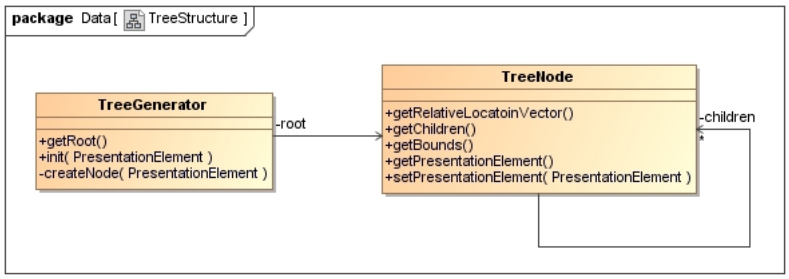

FIGURE 17. The TreeGenerator initializes the tree and knows its root node.

4.3.2. *Create Presentation Group.* Based on this data structure a new action should be created to group a finite number of probably nested Properties in a new PresentationGroup. The only restriction had to be that all elements had the same parent.

To implement this functionality, different methods of other classes had to be used:

At first, a new PresentationGroup and its Property (see page 5) had to be created. MagicUWE provided a method to create a PresentationGroup and to get its Property for further use, but for creating the according PresentationElement, both – the class and the Property – were necessary. Therefore an extended method was created in the DrawPropertyAction class, called `createNewPGAndGetProperty()`.

The next method to be used was in an API class called PresentationElementsManager. This manager offers a method to create a new PresentationElement. Since not only the new PresentationGroup, but all altered elements had to be recreated, this method was fundamental and frequently used.

Since our method was based on the TreeStructure, a problem occurred when more than one element had to be shifted into a new group. A new tree was created for every selected element, so there was no unified root. To solve this problem, a Node for the newly generated PresentationGroup was created, adding all the other trees as equivalent children.

```
1 Vector<TreeNode> nodesForFatherElementsForNewPG = new Vector<TreeNode>();
2         for (PresentationElement pE : elementsForNewPG)
3         {
4                 treeGenerator.init(pE);
5                 nodesForFatherElementsForNewPG.add(treeGenerator.getRoot());
6         }
7
8         TreeNode fatherNodeOfAll = new TreeNode(newPGPresentationElement,
              nodesForFatherElementsForNewPG);
```

Listing 4. Connecting all trees as children of the new PresentationGroup

After this workaround, the method `insertChildrenAfterChangesOnpG()` in the SharedFunction class is called. This method recursively creates a new PresentationElement for every given Node with the given parent, and updating its reference without changing the saved bounds.

At last, all elements had to be positioned. In order to preserve most of the former layout, only the direct children of the root element were auto aligned with the help of the AutoAlignAction. This action provides a special constructor for this call, bypassing different checks and preparations for normal usage.

All nested elements regained their relative position to their parent element, which was saved in the according TreeNodes.

4.3.3. *Delete Presentation Group.*

As well as creating new groups, deleting them without losing their children would be a big assistance in modeling presentation diagrams.

After setting up the TreeStructure, this action was relatively easy to implement. The short algorithm gets the parent of the group to be deleted and creates a TreeStructure for all its children.

Afterwards, the `insertChildrenAfterChangesOnPG()` method is called, giving the children and the parent as parameters.

This time, it was not necessary to use auto aligning. By getting the Node's old bounds, all elements could be recreated at their last location, granting an optimal usability.

4.4. Validate. Since the UML model does not get updated after rearranging elements in a Presentation Diagram, problematic discrepancies could occur when UWE's code generation plugin UWE4JSF (see chapter 2 section 1) was used. Therefore it was necessary to check whether the structure of model and diagram is the same. Furthermore, the model should be adjusted optionally, if these discrepancies were found.

To validate the structure, the order in the UML model was compared to the order in the diagram. The order of the model could be accessed by a API method called `getOwnedAttributes()`, while the order in the diagram could be determined by the same method as in section 4.1. The implemented algorithm validates the structure by comparing the successor of each direct child of the selected element in the model and in the diagram.

If there are discrepancies, both orders will be displayed by colored numbers to the left of each element. Those elements that do not match will be highlighted by a green frame.

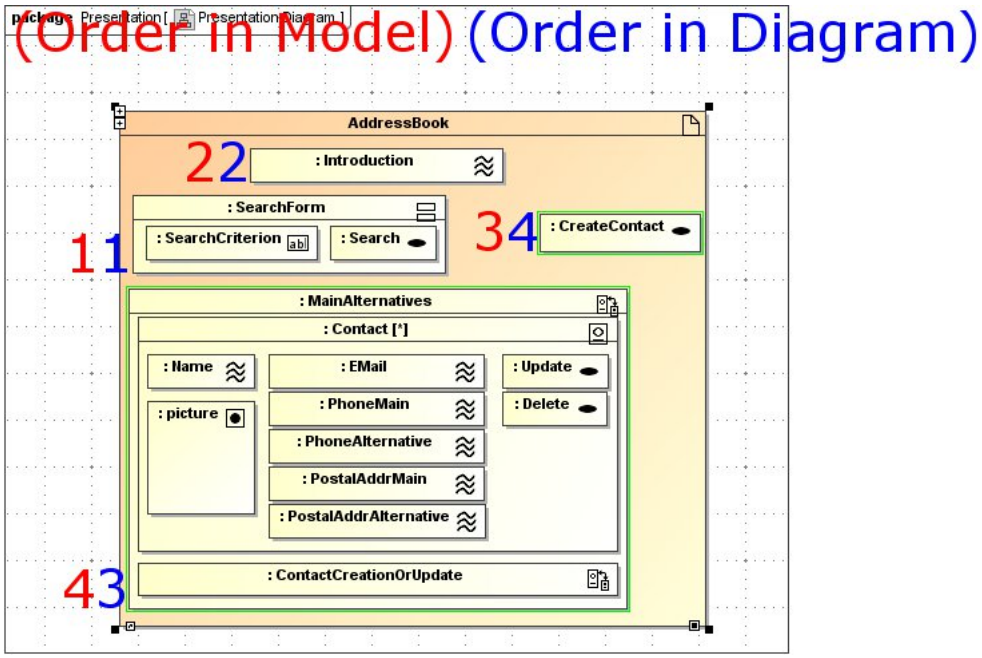

FIGURE 18. If the ValidateAction finds discrepancies, they will be displayed like this.

In the meantime a pop up asks the user whether the model should be aligned according to the diagram. If so, the Collection ownedAttributes of the selected element's model class will be emptied and filled again in the correct order afterwards.

5. Shared Functions

The larger your project, the larger is the possibility to have redundant code. To avoid this problem from the start, a library class had to be implemented. This class should provide all methods and algorithms that were used by more than one action.

In the TidyDiagram plugin, the SharedFunctions is this library class. This library provides different methods for searching PresentationElements, moving them and getting their location in the diagram. Every single action uses a method of this library, the CollisionAction is even only a wrapper for the according method in this class.

Method	**Is used by...**
`RecursiveAddChildrenToList()`	AutoAlignAction ValidateAction
`CheckAndHandleCollisions()`	AutoAlignAction CollisionAction CreatePresentationGroupAction
`Adjust()`	AutoAlignAction CollisionAction
`sortElementsByGUILocation()`	AutoAlignAction ValidateAction
`InsertChildrenAfterChangesOnPG()`	CreatePresentationGroupAction DeletePresentationGroupAction

Since most of the methods have been explained before, the last two should be addressed briefly, too.

The first and smaller method is `adjust()`. This quite simple method resizes and relocates the given PresentationElement due to the given rectangle. While the OpenAPI offers a library method to perform this reshaping, it was not possible to use it, since it caused some severe updating problems after reshaping more than one element.

The second method `recursiveAddChildrenToList()` fetches all PartViews, which represent a Property, up to a given depth relating to the current element. This method is called to get all elements that should be aligned by the AutoAlignAction or be validated by the ValidateAction.

6. Auxiliary Functions

Like mentioned before, one of the main problems was the insufficient documentation of some parts of the OpenAPI and furthermore its restrictions in the GUI domain. Without any knowledge of the different kind of Views or even of their existance, the first steps of development produced no results at all.

To give an example, the OpenAPI method `getPresentationElements()` is described in the javadoc as follows: "Returns all children of this element"
From one perspective, this is true, because this element returns the different and never mentioned parts of the current element, like the HeaderView (chapter 2, section 2.3). But the more intuitive way to understand this method, especially if these Views were never mentioned, is to get all children of our current element that have model elements.

After some failed attempts, it was obvious that more information was necessary to understand the inner mechanics of MagicDraw. To get this knowledge, MagicDraw's data structure had to be made visible. Therefore a new class was created, in order to print this structure into a XML file.

To get as much information as possible, this class was extended to create two files simultaniously if necessary. In the end, AuxiliaryFunctions was able to print all PresentationElements of the active diagram, all selected elements or any given String in up to two files, given maximal flexibility to understand not only the data structure, but also the meaning of the different Views. Finally, it provided the ability to crosscheck any rearranging attempts in the hierarchy.

Since this functionality was not a part of the requirements and especially useless to the user, there is no way to turn it on during runtime. To switch it on, the variable "inTestMode" in the AutoAlignAction's standard constructor has to be be set to "true". After this is done, the class will print all elements of the current diagram in a XML file called DiagramStructure. This file can be found in the folder 'TidyDiagram_temp', located in your operating system's user folder. Every alteration of its functionality has to be hard coded, the javadoc will provide further information if necessary.

CHAPTER 4

Summary

TidyDiagram offers a valuable set of methods to arrange and restructure elements in UWE's Presentation Diagram. These will simplify the daily work of developers. Before achieving this, a lot of research was necessary to understand the underlying data structure of MagicDraw's diagrams. Nearly one third of the time spent was consumed by compensating the flaws in the documentation. Nonetheless, the OpenAPI provides a powerful tool-set for developers, making it possible to design plugins to their own specifications. Being one of them, TidyDiagram extends the set of UWE tools, taking another step towards a user friendly and easy to use program.

Bibliography

[1] UWE – UML-based Web Engineering
Ludwigs-Maximilians-Universität, Programming and Software Engineering, http://uwe.pst.ifi.lmu.de, accessed April 2010

[2] Migration und Erweiterung des MagicDraw-Plugins MagicUWE zur Entwicklung von Web-Anwendungen
Marianne Busch, 31.03.2009

[3] NoMagic Community Forum
NoMagic Inc., https://community.nomagic.com, accessed July 2010

[4] MagicDraw OpenAPI user guide
NoMagic Inc., 2010

[5] Profile helfen bei der Arbeit mit UML
Stefan Queins, Chris Rupp, http://www.computerwoche.de/knowledge_center/software/572838, accessed April 2010

Hiermit versichere ich, dass ich die vorliegende Projektarbeit selbst-ständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Gröbenzell, den 16.08.2010

.....

(Unterschrift des Kandidaten)